2020

Towards understanding the challenges faced by machine learning software developers and enabling automated solutions

Md Johirul Islam
Iowa State University

Towards understanding the challenges faced by machine learning software developers and enabling automated solutions

by

Md Johirul Islam

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Hridesh Rajan, Major Professor
Gianfranco Ciardo
Gurpur Prabhu
Jin Tian
Anuj Sharma

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

DEDICATION

To my teachers, family and friends, who made me realize the real purpose of education.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. I would like to thank Dr. Hridesh Rajan for his guidance, patience, and support throughout this research and the writing of this thesis. I am thankful to the US National Science Foundation for financially supporting this project under grants CCF-15-18897 and CNS-15-13263. I would like to thank my committee members Dr. Gianfranco Ciardo, Dr. Gurpur M. Prabhu, Dr. Jin Tian, and Dr. Anuj Sharma for their efforts and contributions to this work. I would like to extend my thanks to all the members of the Laboratory of Software Design for offering constructive feedback and timely suggestions during research. I would like the convey my special thanks to Hamid Bagheri, Gian Nguyen for helping me in data collection and manual labeling. I would also like to thank Dr. Hoan Nguyen to suggest improvements in my works and helping me in editing the papers.

The majority of this draft is adopted from the prior peer-reviewed papers in top tier software engineering venues. Chapter 3 is based on our paper [2] which is currently under revision for resubmission as per the reviewers feedback. This thesis contains the updated revision of the draft. Chapter 4 is based on our accepted paper [3] at ESEC/FSE 2019. Chapter 5 is based on accepted paper [4] at ICSE 2020. Chapter 6 is based on our draft submitted to FSE 2020 and is being revised for resubmission. This thesis contains a revised version of the draft addressing the most major comments from the reviewers.

In this thesis, I do not include some of the other papers that I worked on during my Ph.D. For example, I got the chance to work on extending Boa [5] and show its effectiveness in the analyses of large scale transportation data, and our paper was accepted at the Journal of Big Data Analytics in Transportation [6]. I also got the chance to collaborate in creating a large scale dataset for analyzing data science programs that have been accepted at the data showcase of MSR 2019 [7]. I also got the chance to work on several projects related to the analyses of ML models, adversarial attacks to ML, and runtime monitoring of Deep Neural Network models [8].

ABSTRACT

Modern software systems are increasingly including machine learning (ML) as an integral component. However, we do not yet understand the difficulties faced by software developers when learning about ML libraries and using them within their systems. To fill that gap this thesis reports on a detailed (manual) examination of 3,243 highly-rated Q&A posts related to ten ML libraries, namely *Tensorflow*, *Keras*, *scikit-learn*, *Weka*, *Caffe*, *Theano*, *MLlib*, *Torch*, *Mahout*, and *H2O*, on *Stack Overflow*, a popular online technical Q&A forum. Our findings reveal the urgent need for software engineering (SE) research in this area.

The second part of the thesis particularly focuses on understanding the Deep Neural Network (DNN) bug characteristics. We study 2,716 high-quality posts from *Stack Overflow* and 500 bug fix commits from *Github* about five popular deep learning libraries *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch* to understand the types of bugs, their root causes and impacts, bug-prone stage of deep learning pipeline as well as whether there are some common antipatterns found in this buggy software.

While exploring the bug characteristics, our findings imply that repairing software that uses DNNs is one such unmistakable SE need where automated tools could be beneficial; however, we do not fully understand challenges to repairing and patterns that are utilized when manually repairing DNNs. So, the third part of this thesis presents a comprehensive study of bug fix patterns to address these questions. We have studied 415 repairs from *Stack Overflow* and 555 repairs from *Github* for five popular deep learning libraries *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch* to understand challenges in repairs and bug repair patterns. Our key findings reveal that DNN bug fix patterns are distinctive compared to traditional bug fix patterns and the most common bug fix patterns are fixing data dimension and neural network connectivity.

Finally, we propose an automatic technique to detect ML Application Programming Interface (API) misuses. We started with an empirical study to understand ML API misuses. Our study shows that ML API misuse is prevalent and distinct compared to non-ML API misuses. Inspired by these findings, we contributed Amimla (Api Misuse In Machine Learning Apis) an approach and a tool for ML API misuse

detection. Amimla relies on several technical innovations. First, we proposed an abstract representation of ML pipelines to use in misuse detection. Second, we proposed an abstract representation of neural networks for deep learning related APIs. Third, we have developed a representation strategy for constraints on ML APIs. Finally, we have developed a misuse detection strategy for both single and multi-APIs. Our experimental evaluation shows that Amimla achieves a high average accuracy of $\sim$80% on two benchmarks of misuses from *Stack Overflow* and *Github*.

CHAPTER 1. INTRODUCTION

Machine Learning (ML) is becoming an essential component of modern software engineering. Moreover, its usage in safety-critical systems like self-driving cars, automatic traffic and flight control, robotic surgery, etc. is on the rise. Software developers use highly abstract machine learning libraries to write ML programs in their tools and applications. A large number of ML libraries are publicly available to support the increasing usage of ML in the development of software. For example, some of the popular libraries are *Caffe* [9], *H2O* [10], *Keras* [11], *Mahout* [12], *MLlib* [13], *scikit-learn* [14], *Tensorflow* [15], *Theano* [16], *Torch* [17], and *Weka* [18]. All these libraries abstract away the details of ML algorithms and provide Application Programming Interfaces (API) to the developers. Therefore, developers can easily write complex ML models using these APIs. This has given rise to the discussions related to the trustworthiness, correctness, robustness and performance of the models, characteristics of bugs in the ML programs, developers challenges, bug fix patterns, etc.

Therefore, it has become an urgent need to study the challenges faced by the developers in developing ML applications to pave the research for solving the ML developers' challenges. To the best of our knowledge, no such studies existed prior to the work in this thesis. So, to fill in this gap we studied the challenges faced by ML developers in using the ML libraries to develop software, the characteristics of the bugs made by the deep learning software developers, the patterns followed by the developers in fixing those bugs, and we also proposed a novel technique to automatically detect API misuses in ML software.

Our key findings and results show that ML developers face most challenges in the model creation stage of the ML pipeline; developing deep learning applications are highly bug-prone with new patterns of bugs; fixing deep learning applications have several new and unique patterns along with new challenges. We also showed that a novel ANN (Abstract Neural Network) based API misuse detector is more effective in ML API misuse detection compared to the state of the art prior works in other domains of software engineering.

1.1 Contributions

In this thesis, we provide several studies to understand the challenges of ML software developers and propose an automatic technique to detect one of the bugs ML developers are likely to make. Our specific contributions are:

In chapter 3, we provide an empirical study to understand the difficulties faced by ML developers in developing ML applications. We used *Stack Overflow* to understand the difficulties by analyzing the discussions for the 10 most popular ML libraries. We studied 3,243 highly-rated posts from *Stack Overflow* manually and conduct thorough intra-library and inter-library analyses to answer five different research questions identifying the challenges faced by the developers.

In chapter 4, we conducted another empirical study to understand the characteristics of bugs made by the developers in developing Deep Neural Network (DNN) software. We have studied 2,716 qualified *Stack Overflow* posts in five different deep learning libraries and done both qualitative and quantitative analyses to answer five different research questions. We have found several unique bug patterns in DNN that needs innovations from software engineering research.

In chapter 5, we studied the patterns of repairing DNN programs by the developers. We have studied the 415 bugs from *Stack Overflow* and 555 bugs from *Github* from chapter 4 and manually analyzed how those bugs are fixed to identify the fix patterns. We also answered five different research questions through detailed qualitative and qualitative analyses. We have reported several newly appearing patterns for fixing DNN applications and we have also identified some new challenges in fixing DNN software.

In chapter 6, we proposed an automatic API misuse detector for ML API misuses made by the developers. We proposed a novel intermediate representation that is very effective in fixing the DNN bugs related to API misuse. We have shown that our approach outperforms the state of the art API misuse detectors in detecting bugs in ML. We have also shown the effectiveness of our approach by fixing some real-world bugs in the repositories by industry leaders like Google and IBM.

We also contributed four different publicly available manually labeled datasets that have the potential to advance the research related to fixing the problems faced by the DNN developers and enable follow up studies.

1.2 Outline

In the next chapter, we provide a detailed literature review on empirical studies conducted for understanding challenges in other domains of software engineering, bug patterns, and program repair patterns in non-ML fields, API misuse, and API misuse detection techniques in non-ML domains. In chapter 3, we provide a large study on the challenges faced by the developers using *Stack Overflow* and we answer several research questions through qualitative and quantitative analyses of data. In chapter 4, we provide a comprehensive study on the bug characteristics made by the DNN software developers. In chapter 5, we identify and study the program repair patterns for DNN programs and describe several unique DNN specific program patterns along with some new challenges. In chapter 6, we propose Amimla, an automatic technique to detect the API misuses by ML programmers. We evaluate Amimla with a state of the art technique and have demonstrated our approach is effective in detecting API misuses by detecting and fixing some API misuses by industry leaders like Google and IBM. Finally, in chapter 7, we provide a conclusion and several new directions of research evolving from this thesis.

CHAPTER 2. RELATED WORK

2.1 Empirical Study Using *Stack Overflow*

Stack Overflow is the widely used platform to study software engineering practice from the developer's perspective. However, existing work has not studied the usage of ML libraries using *Stack Overflow*. Meldrum *et. al.* [19] studied 266 papers using *Stack Overflow* platforms to show the growing impact of *Stack Overflow* on software engineering research. Treude *et. al.* [20] did manual labeling of 385 questions to manually classify 385 questions into 10 different categories (how-to, discrepancy, environment, error, decision help, conceptual, review, non-functional, novice, and noise) to identify the question types. This study is useful to learn the general categories of questions asked by developers.

Kavaler *et. al.* [21] used *Stack Overflow* data to study the queries on APIs used by Android developers and showed the correlation between APIs used in producing Apps in the market and the questions on APIs asked by developers. Linares-Vásquez *et. al.* [22] studied the effect of the changes in Android API on the developer community. They used the discussions arising on *Stack Overflow* immediately after the API is changed and the behavior of the API is modified to study the impact of the change among the developers.

Berral-García *et. al.* [23] studied machine learning-based algorithms, approaches, execution frameworks and presented a brief discussion of some libraries used in machine learning. Barua *et. al.* [24] studied *Stack Overflow* posts and used LDA topic modeling to extract topics to study the trend of different topics over time.

Rebouças *et. al.* [25] studied the usage pattern of swift programming language among developers using *Stack Overflow* data.

Schenk *et. al.* [26] studied the geographical distribution of usage and knowledge of different skills using *Stack Overflow* posts and users data. Stanley *et. al.* [27] proposed a technique based on the Bayesian probabilistic model to predict the tags of a *Stack Overflow* post. McDonnel *et. al.* [28] presented a study of API stability using *Stack Overflow* data and as a test case, they used the Android Ecosystem. Baltadzhieva

et. al. [29] proposed a technique to predict the quality of a new *Stack Overflow* question. Joorabchi *et. al.* [30] studied the challenges faced by computer science learners in different topics and subjects using the *Stack Overflow* data.

2.2 Study of Challenges in ML

The authors in [31, 32] have explored some major reasons of technical debts in ML and claimed that the quick advances in ML are not coming out for free. The authors described that these technical debts are caused by complex models, data dependencies, feedback loops, and ML anti-patterns. The authors described these observations based on their experience and discussion with the researchers and developers at Google.

On the other hand, in chapter 3 we conduct a large-scale study using the problems discussed by the developers in a well-trusted Q&A forum. The difficulties faced by the developers using different libraries and creating different types of ML models enhance the coverage of this study. Our findings also comply with the discussions on technical debts made by [31, 32] on model complexity and data dependency.

Eric *et. al.* [33] studied software from Google and built a point-based testing framework that checks the data, features, and model development process. Our key findings also suggest that model creation and data adaptation are the most prevalent challenges across all ML libraries we have studied. Our study has focused on other aspects of ML as well e.g., challenges faced in different stages of ML pipeline, commonalities and variabilities of the difficulties faced by the developers in the ten ML libraries, and the evolution of different challenges over the years, etc.

A more relevant study has been conducted by [34] that focuses on the problems faced by developers in Microsoft in different stages of the ML pipeline. This work focus on processes and pipelines in ML. On the contrary, we focus on APIs and cross library comparison.

Their findings show that *data availability, collection, cleaning, and management* and *Model evolution, evaluation, and deployement* are some key problems faced by the professionals. However, the analyses presented in this thesis supports the difficulties observed at the industry level by [34] as well present a number of new findings e.g., type mismatches are prevalent, shape mismatch mostly occurs in Keras, most issues in ML code leads to failure, parameter selection, choice of loss function are a difficult task for most

of the ML developers, the commonality and variability of the 10 libraries in terms of issues found, etc. One other major contrast with [34] is, the authors in [34] studied ML pipeline and workflow using a survey, but we study the ML libraries using Q&A forum.

2.3 Empirical Study of Bugs in Non-ML

There are some empirical studies focused on specific types of bugs. Lu *et. al.* [35] studied real-world concurrency bug characteristics. Gao *et. al.* [36] conducted an empirical study on recovery bugs in large-scale distributed systems. API changes problems was studied by [37–40]. Our work focuses on the bugs in the usage of deep learning libraries.

Other prior work that have studied *Stack Overflow* posts, e.g. [19, 21, 22, 24], have not focused on deep learning software.

Pan, Kim, and Whitehead [41] have studied seven Java projects to discuss the bug fix patterns in these projects. They have also proposed a classification scheme in categorizing the bug fix patterns in Java. The classification includes 9 broad categories and a total of 26 lower-level categories. This prior research suggests that the IF-related and Method Call (MC) related bugs are most frequent. In DNN bug fix strategies, the MC and sequence addition or deletion related bug fix pattern is present. We do not find any evidence of other bug fix strategies in DNN and that has inspired us to derive a classification scheme using the open coding approach to classify the DNN bug fix patterns.

Programming bugs are well-studied in software engineering. There is a rich body of empirical studies on bugs, e.g. [35, 42–47] and bug repair, e.g. [41, 48–50]; however, these works have not studied DNN bugs and repairs that have their own set of unique challenges [2, 3, 51, 52].

2.4 Empirical Study of Bugs in DNN

The closest related work on deep learning bug characteristics is by Zhang *et al.* [51] who have investigated bugs from deep learning applications built on top of Tensorflow. They collected 500 *Stack Overflow* posts and filtered them to study 87 posts and selected 11 Github projects to include 82 commits with 88 bugs using keywords e.g., bug, fix, wrong, etc. In contrast, we studied a cross-section of five deep learning

libraries with different design constraints, using 555 bugs from GitHub and 415 *Stack Overflow* posts, that allowed us to draw interlibrary observations.

Zhang *et al.* have studied the bugs and have categorized them into 7 types of bug/root causes and 4 types of impacts/symptoms. Our work expanded the study to include bug types from literature and categorizes the bugs into 11 bug types, 10 root causes, and, 7 impacts. In term of results, our study both confirms what was known as a small scale and produces new knowledge e.g., correlating antipatterns with bugs [53].

Thung *et al.* [52] studied three machine learning systems, Apache Mahout, Lucene, and OpenNLP, and manually categorize the bugs into different categories. They focused on bug frequencies, bug types, the severity of the bug, bug-fixing duration, bug-fixing effort, and bug impact. Different from them, we focus on bug types, bug root causes, and bug impact of five deep learning libraries which are Tensorflow, Keras, Torch, Caffe, and Theano.

2.5 Study of Fixing DNN Bugs

Zhang *et al.* [51] have studied bug patterns in *Tensorflow* using both *Github* and *Stack Overflow*. They have discussed the new patterns and characteristics of the bugs by *Tensorflow* users to write DNN applications. They have also discussed the three new challenges in detecting and localizing these bugs. The study was limited to *Tensorflow* and also does not discuss the bug fix patterns. We generalize to a number of deep learning libraries and identify the new patterns of fixing the bugs in DNN software. We also discuss the new challenges in fixing these bugs. We have discussed three new challenges in fixing DNN bugs.

Sun *et al.* [54] studied the issues in 3 ML libraries *scikit-learn*, *Caffe*, and *paddle* to understand the bug fix patterns in these libraries. However, we study the DNN models created using DNN libraries. Our findings do not have any commonalities.

Zhang *et al.* [55] studied 715 *Stack Overflow* bug related posts for TensorFlow, PyTorch, and Deeplearning4j to classify the questions into 7 different categories and built an automated tool that categorizes questions based on the frequently found words from each category and computing the tf-idf value with respect to the keywords. Also, the authors have studied the challenges of answering the question in *Stack Overflow*

by calculating the response time for each category and have found 5 categories of root causes for the bug related posts. Whereas, our study has been on the bug fixing strategies for 5 DNN libraries.

Pham *et al.* [56] studied three deep learning (DL) libraries *Tensorflow*, *CNTK*, and *Theano* to localize and detect deep learning bugs using cross-validating different backend i.e., *Tensorflow*, *CNTK*. In contrast, our work studied five DL libraries, using bug-fix commits from *Github* and *Stack Overflow* posts, that allowed us to draw interlibrary observations of the fixing strategies. Also, our work expanded the study to include deeper discussion about each fix pattern, the common challenges developers face while fixing these bugs, and how fixing bugs introduces a new bug.

2.6 Study of API-misuse Classification

Classification of API-misuse from different perspectives and domains have been done previously. Classification of software defects by IEEE served as the basis for IBM's orthogonal defect classification (ODC) as discussed in [57]. The defect types include conceptual program elements which are *function*, *check*, *documentation*, *assignment*, *algorithm* and *violation type*. More recently, [58] classified API-misuse for Java programming language.

The classification includes API-misuses like *method calls*, *conditions*, *iterations*, and exception handling. None of them support misuse detection for ML APIs.

2.7 API Misuse Detection

Amimla is the most closely related to static approaches for API-misuse detection. They typically mine frequent API usages, which are considered as good usages or usage patterns, and use them to detect misuses which are usages violating those patterns.

Ganter *et. al.* [59] extract call sequences from programs to perform formal concept analysis to analyze pairwise temporal properties of API calls as described by [60]. There has been works to infer temporal properties of API calls described in the works by [61–66]. There have also been works to mine and model program as itemsets as described by [67–69] and infer pairwise programming rules by frequent itemset mining. [70] mines programs from *Github* and guard conditions against the predicates of the APIs.

GROUMiner [71] uses graphs to represent API usages and patterns, and detect misuses. Amimla uses mining for building the API canonical forms. Different from those works, we manually create the constraints of APIs from documentation and discussions of API usages. More importantly, Amimla models the network layers in neural network APIs with ANNs and API sequences as ML pipelines. Amimla uses ANNs to detect incompatibility between layers, and ML pipelines to detect incompatibility between stages.

2.8 Model Testing

Recent work has studied methods to validate ML models. For example, [72] used concolic testing to detect safety problems in deep neural networks. Specifically, they increased the test coverage by formalizing coverage criteria for DNNs. [73] used SMT to create a verification framework for feed-forward-multi-layer neural networks. This work supports deep learning developers by validating image classification decisions. [74] created a technique to verify the reachability properties of deep neural networks based on the simplex method. They have extended the simplex method to detect the reachable state's models with ReLU activation function. [75] focuses on the robustness of DNNs by studying a generic reachability problem for feed-forward DNNs. Compared to these works, the focus of Amimla is on ML API misuse detection.

CHAPTER 3. WHAT DO DEVELOPERS ASK ABOUT ML LIBRARIES? A LARGE-SCALE STUDY USING STACK OVERFLOW

3.1 Introduction

Machine learning (ML) is becoming an essential computational tool in a software developer's toolbox for solving problems that defy traditional algorithmic approaches. Software developers are fulfilling this need by development and refinement of a number of new ML libraries [76]. Recently it has also been suggested that ML can introduce unique software development problems [31–33]. However, we do not yet know about the problems that users of ML libraries face and those that they choose to ask about publicly.

Prior work has shown that studying question and answer (Q&A) forums such as *Stack Overflow* can give significant insights into software developer's concerns about a technology [20–22, 24–30, 77–81], but has not focused on ML libraries. More details of related work are discussed in chapter 2.

This work presents a study of the problems faced by developers while using popular ML libraries. Our study also leverages the posts on *Stack Overflow*. Since 2015, there has been growing interest and significant increase in ML related questions and distinct users making *Stack Overflow* a representative source of dataset for our study. We selected 10 ML libraries to study, identified by a survey [76] and confirmed by counting the number of posts on *Stack Overflow* related to those libraries. These libraries are *Caffe* [9], *H2O* [10], *Keras* [11], *Mahout* [12], *MLlib* [13], *scikit-learn* [14], *Tensorflow* [15], *Theano* [16], *Torch* [17], and *Weka* [18].

Caffe [9] is a deep learning library for Python and C++. ***H2O*** [10] is a deep learning library for Java, R, Python or Scala, and its key feature is to provide a workflow-like system for building ML models. ***Keras*** [11] is a deep learning library for Python whose key feature is to provide higher-level abstractions to make creating neural networks easier. *Keras* also uses *Tensorflow* or *Theano* as the backend. ***Mahout*** [12] is aimed at providing scalable ML facilities for Hadoop clusters. ***MLlib*** [13] is aimed at providing scalable ML facilities for Spark clusters. ***scikit-learn*** [14] is a Python library that uses *Tensorflow* or *Theano* as

the backend. This library provides a rich set of abstract APIs [82] to hide the complexity of ML from the user in an effort to make ML features widely accessible. ***Tensorflow*** [15] provides facilities to represent a ML model as data flow graphs. ***Theano*** [16] and ***Torch*** [17] are aimed at scaling ML algorithms using GPU computing. A novelty of *Theano* is that it provides some self-verification and unit testing to diagnose some runtime errors. ***Weka*** [18] is an ML library for Java. It provides API support for data preparation, classification, regression, clustering, and association rules mining tasks and a GUI for making models easier.

All in all, this set is both representative and provides variety. We selected a total of 3,243 highly-rated *Stack Overflow* posts for this study. A team of three Ph.D. students, with experience in coursework on AI and ML, and using ML libraries, independently read and labeled each of the posts producing 9,849 labels that were then compared for consistency producing 177 conflicting labels on 177 different posts. All of these conflicts were resolved using mediated, face-to-face conflict resolution meetings between all three participants. We then performed statistical analysis and a study of the data to answer the following research questions:

RQ1: Difficult stage Which stages are more difficult in an ML pipeline? Figure 3.1 shows stages in a typical ML pipeline.

RQ2: Nature of problems Which problems are more specific to the library and which are inherent to ML?

RQ3: Nature of libraries Which libraries face problems in specific stages and which ones face difficulties in all stages?

RQ4: Consistency Did the problems stay consistent over time?

The remainder of this chapter describes our study and results and makes the following contributions: (1) a labeled and verified, a dataset of 3,243 ML library-related Q&A on *Stack Overflow*, (2) a classification scheme for ML-related Q&A, (3) an intra-library analysis to identify the strengths and weaknesses of ML libraries, and (4) an inter-library analysis to identify relative strengths and weaknesses.

3.2 Methodology

Our study uses Q&A posts on *Stack Overflow*, a popular platform used by developers. Our first step was to find the total number of questions asked about all the ML libraries highlighted by some recent

Table 3.1: Numbers of posts having different score (S) about ML libraries. The bold column represents selected posts. $S = |N_U| - |N_D|$ where $|N_U|$ is the number of upvotes and $|N_D|$ is the number of downvotes.

Library	S ≥ 0	S ≥ 1	S ≥ 2	S ≥ 3	S ≥ 4	S ≥ 5
Caffe [9]	2,339	1,320	620	318	192	**132**
H2O [10]	771	452	167	73	34	**17**
Keras [11]	5,708	3,323	1,751	953	568	**367**
Mahout [12]	1,186	610	293	160	103	**48**
MLlib [13]	1,688	929	498	272	173	**119**
scikit-learn [14]	9,246	5,302	2,898	1,759	1,188	**856**
Tensorflow [15]	21,115	10,109	4,962	2,769	1,827	**1,334**
Theano [16]	2,332	1,341	711	421	265	**192**
Torch [17]	1,226	640	312	161	91	**61**
Weka [18]	2,512	1,216	568	293	181	**117**
Total	48,123	25,242	12,780	7,179	4,622	**3,243**

surveys [76, 83, 84]. Out of these, we selected 10 popular ML libraries for the study as shown in Table 5.1. We excluded the other five libraries because the numbers of questions about them were too few (less than 20).

On *Stack Overflow*, each question is rated by the community. The score of a question is computed as $S = |N_U| - |N_D|$ where $|N_U|$ is the number of upvotes and $|N_D|$ is the number of downvotes. The higher score is an indicator of the higher quality of the question, which has been used by prior works such as [19]. Table 5.1 shows the entire distribution of the questions for each library based on the score of S.

We selected questions with the score of 5 or higher (bold column in Table 5.1) to focus on high-quality questions while keeping the workload of manually labeling each question manageable. Note that in *Stack Overflow* duplicate questions are not allowed, and do not receive upvotes because moderators mark them as duplicate. So, duplicate questions are eliminated when we use the score as quality metric.

Next, we manually classified each *Stack Overflow* question into categories to study them further. We first discuss the classification of categories and then our labeling process.

3.2.1 Classification of Questions

We classify the questions in *Stack Overflow* into several categories. First, we classify the questions into two top-level categories based on whether the question is related to ML or not. Questions related to instal-

lation problems, dependency, platform incompatibility, Non-ML APIs, overriding the built-in functionality, adding custom functionality fall into Non-ML category as shown in Figure 3.2. We classify the ML-related questions into six categories based on the stages of a typical ML pipeline [1], also reproduced in Figure 3.1. Among those seven stages, data collection is out of the scope of this study because ML libraries do not provide this functionality, which leaves us with six categories. These six categories are further divided into different subcategories. To find these subcategories one of the Ph.D. students with ML expertise first studied 50% of posts and created the subcategories using open coding scheme adapted from earlier works [85–87]. These labelings were not exposed to the manual raters as discussed later. The goal of this step was to find the coding scheme needed to label the difficulties using an open coding approach. Then these subcategories were sent to three ML experts for review. The three ML experts are faculties who are actively involved in Artificial Intelligence and Machine Learning research. Based on the review of the experts, the subcategories were improved and the process continued until an agreement with ML experts was reached. Once, we reached an agreement on the classification scheme, each rater rated all the posts using this classification scheme.

Then, we have reported the results for each library in terms of the number of posts. We have utilized the difficulty of each category as the number of posts labeled as that particular category. To validate that the number of posts is a representative metric of the difficulty in *Stack Overflow*, we have studied a prior work [88] that has identified the different bugs found in Java and Python. Then, we have counted the posts with tags i.e., for Java with performance-related issues, we have counted the number of posts under the tag ‘performance‘ and ‘java‘. We have found that there is a correlation between the number of posts about an issue and the difficulty faced by the developers. According to the authors, the top 3 problems faced by the Java and Python developers are Performance, Memory, and Security respectively. We have found that the count of posts also reflects this difficulty in Java and Python (Performance (3260)$>$ Memory (2093)$>$ Security (1870)). Our study indicates that the number of posts can be utilized as a representation of the challenges found in the prior study. The full classification is shown in Figure 3.2. We keep the Non-ML as a separate category as it covers a major share of code in ML projects. The authors in [31] discussed that only 5% of the code in an ML project is related. Next, we describe its categories.

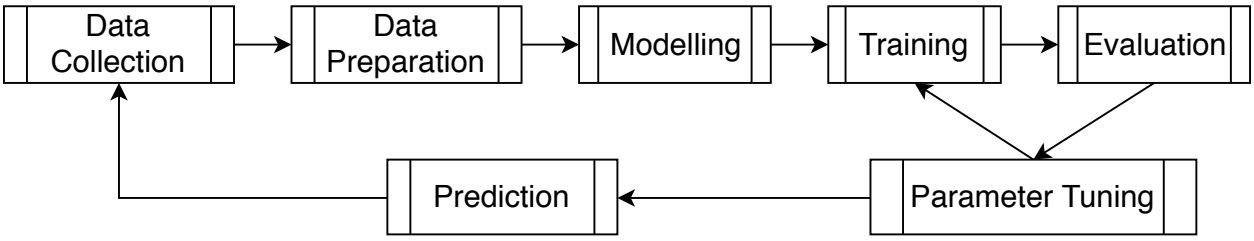


Figure 3.1: Stages in a typical ML pipeline, based on [1].

3.2.1.1 Data Preparation

This top-level category includes questions about converting the raw data into the input data format needed by the ML library.

Data adaption. Questions under this subcategory are about reading raw data into the suitable data format required by the library. Data reader provided by the library usually provides this functionality. Questions about converting data, encoding, etc., also fall under this subcategory.

Featuring. Questions under this subcategory are about feature extraction and selection. *Feature extraction* is a process to reduce the dimensionality of the data where existing features are transformed into a lower-dimensional space. *Feature selection* is another strategy of dimensionality reduction where informative features that have an impact on the model are selected.

Type mismatch. Type mismatch happens when the type of data provided by the user doesn't match the type required by the ML API. For example, if an API needs `floating point` data as input but the client provides a `String` then the ML API raises an exception.

Shape mismatch. Shape mismatch occurs when the dimension of the tensor or matrix provided by a layer doesn't match the dimension needed by the next layer. These kinds of errors are very common in deep learning libraries.

Data cleaning. Data cleaning phase, sometimes also called data wrangling, includes removal of null values, handling missing values, encoding data, etc. Without proper data cleaning the training may throw exceptions, and accuracy may be suboptimal.

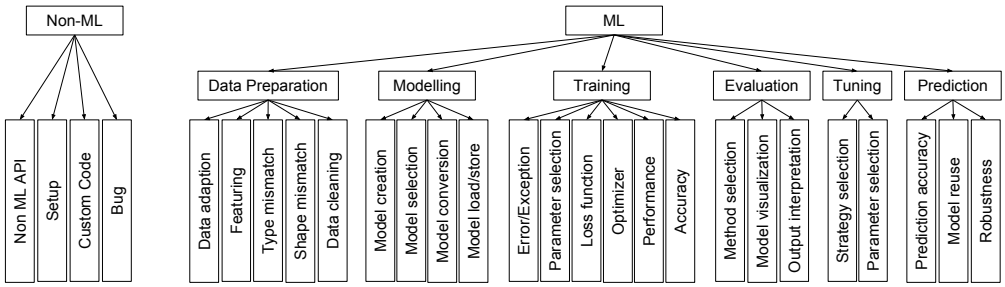


Figure 3.2: Classification used for categorizing ML library-related Stack Overflow questions for further analysis

3.2.1.2 Modelling

The subcategories of this category include:

Model selection. This subcategory includes questions related to the choice of the best model and choice of the API version (e.g. whether to chose SVM or decision tree).

Model creation. This subcategory includes questions related to creating the ML model using the APIs.

Model conversion. This subcategory includes questions related to conversion of a model trained using one library and then using the trained model for prediction in an environment using another library. For example, a model trained in *Torch* can be used for further training or prediction using *Theano*.

Model load/store. This subcategory contains questions about storing models to disk and loading them to use later.

3.2.1.3 Training

The subcategories of this category include:

Error/Exception. Questions about errors faced by users in the training phase fall into this subcategory. The errors may appear due to various reasons. If the errors are due to shape mismatch or type mismatch we put them into the data preparation category. Otherwise, all errors are placed into this subcategory.

Parameter selection. Some frameworks have optional parameters, and developers have to choose appropriate values for these parameters and also pass relevant values to the required parameters. Questions related to these problems fall into this subcategory.

Loss function. Questions related to choosing and creating loss functions fall into this category, e.g., whether to use cosine distance.

Optimizer. Questions related to the choice of optimizer are placed into this subcategory, e.g., should I pick Adam or AdaGrad?

Performance. In this subcategory, questions related to long training time and/or high memory consumptions are placed.

Accuracy. Questions related to training accuracy and/or convergence are placed into this subcategory.

3.2.1.4 Evaluation

The subcategories of this category include:

Evaluation method selection. In ML related problems, a model needs to be evaluated against different techniques and the choice of the technique has an impact on both achieving the desired performance and the intent of the developer [89] e.g. "*which of the eight APIs for eight different types of validations in scikit-learn, namely KFold, LeaveOneOut, StratifiedKFold, RepeatedStratifiedKFold, RepeatedKFold, LeaveOneGroupOut, GroupKFold and ShuffleSplit, should be used?*"

Visualizing model learning. The developers sometimes need to visualize the behavior of the model to get a better understanding of the training process and also to know the effects of evaluation on the change of loss function and accuracy. Those questions are placed in this subcategory.

3.2.1.5 Hyper-parameter Tuning

Hyperparameter tuning is used to improve the model's performance. The values of hyperparameters affect model accuracy. For example, a bad learning rate may cause a model to learn poorly and give low accuracy. The subcategories of this category include:

Tuning strategy selection. Questions about choosing among APIs for different tuning methodologies are placed into this subcategory. For example, one poster wondered whether they should use the grid search or randomized search or parameter sampling for parameter tuning in *scikit-learn*?

Tuning parameter selection. This subcategory covers discussions related to the selection of parameters for tuning. Some parameters may not affect the model accuracy other than increasing the training time while some might have a significant effect on the accuracy. For example, the following code from a post is trying to tune the kernel and C parameter of the ML algorithm to find the best combination from values given at line 4.

```
1 from sklearn import svm, datasets
2 from sklearn.model_selection import GridSearchCV
3 iris = datasets.load_iris()
4 parameters = {'kernel':('linear','rbf'), 'C':[1,10]}
5 svc = svm.SVC()
6 clf = GridSearchCV(svc, parameters)
```

3.2.1.6 Prediction

After the model is trained and evaluated, the model is used to predict new input data. Questions in this top-level category are about problems faced by the developers during prediction and include the following subcategories.

Prediction accuracy. Questions related to prediction accuracy, e.g. due to overfitting, are placed into this category.

Model reuse. Developers might have difficulty in reusing existing models with their datasets for prediction to make use of the state of the art models from well-known providers.

Robustness. Questions in this subcategory are about the stability of the models with slight changes, possibly noise, in the datasets.

3.2.2 Manual Labeling

Manual labeling of the Q&A dataset was the most important (and time-consuming) step before our analysis. To decrease the bias in manual labeling we recruited three participants and used inter-rater agreement to provide a common label for each post. Posts are questions with unique id's. However, for labeling, the raters were asked to review the relevant answers with positive scores as well for understanding the problem

well. So when we talk about studying posts in the thesis, we mean studying both the questions and the answers.

Each participant had coursework in both AI and ML and had experience using ML libraries to solve problems. Each participant labeled all the questions producing 9,840 labels. The labels from the raters were mutually exclusive. We have used *Stack Overflow* tags to filter the questions related to a particular library. So a question having tag scikit-learn discusses programming problems related to the scikit-learn only.

Participant Training. Before the labeling, the participants were provided with the classification shown in Figure 3.2. Then, a training session was conducted where each (sub) category was discussed and demonstrated using examples.

Labelling Efforts. First, each participant gave each question one of the labels from top-level categories namely Non-ML, Data Preparation, Modelling, Training, Evaluation, Tuning, Prediction. Then, (s)he assigned a subcategory.

Reconciling Results. After collecting labels separately from each participant, a moderator then compared them. If there was an inconsistency between participants for a question, the moderator created an issue in a repository for resolution. Among all 3,243 questions, 177 (5%) needed further discussion.

Then, the three participants had two in-person meetings to discuss those 177 questions. The participants read the questions carefully again and voted individually. If the votes matched we accepted those as resolved, otherwise, participants discussed the reasons behind choosing a label and tried to achieve consensus. In most cases, the opinions differed due to the ambiguous nature of the questions. For example, for a question asking about suboptimal accuracy, it was difficult to say from the question without further exploration whether it is talking about accuracy in the prediction stage or accuracy in the training or evaluation stage. We resolved these types of questions by a careful reanalysis of the Q&A text.

We measured the inter-rater agreements using Cohen's kappa coefficient (κ) as shown in Figure 3.3a. It measures the observed level of agreement between raters of a particular set of nominal values and corrects for agreements that would appear by chance. The interpretation of κ's values is shown in Figure 3.3b. From Figure 3.3a, we see that the kappa coefficient between all the raters involved in the labeling process is more than 0.9 indicating perfect agreements. We also computed the Fleiss coefficient [90] which is widely used

	R1	R2	R3
R1	1.00	0.94	0.92
R2	0.94	1.00	0.91
R3	0.92	0.91	1.00

(a) Kappa coefficients (κ).

0.00–0.20	slight agreement
0.21–0.40	fair agreement
0.41–0.60	moderate agreement
0.61–0.80	substantial agreement
0.81–1.00	perfect agreement

(b) Interpretation of κ value.

Figure 3.3: Cohen's kappa coefficients for labeling process.

for finding IRR between more than 2 raters. The Fleiss coefficient was 90.68% indicating a perfect level of agreement. We computed all the IRR coefficients based on the ratings before the discussion for agreement. After the discussion for reconciling the conflicts in the presence of a moderator, the agreement level was 100%.

3.2.3 Threats to Validity

Internal Validity In the manual labeling, a threat can be due to the possibility that labeling could be biased. We mitigate this threat by using 3 raters and resolving the conflicts via in-person meetings. The inter-rater reliability coefficient shows that there was a perfect level of agreement between raters.

The possibility of missing relevant posts can also be a threat. We mitigate this bias by collecting the tags that are relevant to a particular ML library. We then collect all the posts containing those tags using *Stack Overflow* API.

Classification of questions in the top-level categories can also pose a threat. To mitigate this threat we use the categorization used and described by practitioners and researchers [1, 13, 91].

Classifying the top-level categories into subcategories can have bias and missing subcategories due to an open coding scheme. To mitigate this threat one Ph.D. student initially studied a subset of posts and came up with the subcategories. Then, three ML experts were consulted and their opinion on the classification was used for multiple rounds of revisions and improvements.

The ML expertise of the raters can affect manual labeling. To mitigate this threat we selected raters who have expertise in ML as well as using the libraries in the study. The raters also study the answers and comments in posts to improve their insights.

External Validity In *Stack Overflow* threat to validity can be low-quality posts [92], and chronological order of posts. To eliminate the quality threat we studied only the posts that have the tag of the relevant library and then only kept the posts that have score $\geq$5. This balanced both the quality and labeling efforts.

The chronological order of the posts can introduce threats as some older posts may be resolved in a later version of the libraries. To alleviate this threat we classify questions that may appear only due to the API versions into the Non-ML category.

An external threat can be the expertise of the programmers asking the questions. If the questions are asked only by newbies then our results aren't as general. To understand this threat, we measured *reputation*, a metric used by *Stack Overflow* to estimate expertise. *Stack Overflow* users above 50 are considered reputable and are allowed to comment on posts. Table 3.2 shows the reputation statistics of developers asking the questions we studied. We have collected the reputation of users in the respective libraries. For example, the reputation statistics for *Keras* will only have the reputation of the users earned by asking and answering questions related to *Keras*. We use tag names to filter out the reputation. For example, if `user1` has a total reputation of 100,000 in different technologies along with 5000 reputations in *Keras*, we only take 5000 as the relevant reputation in *Keras*. Our result shows that both the mean and median reputations are high ($\geq$ 10 as per *Stack Overflow* policy [93]) that indicates the expertise of the programmers asking the questions is not a threat to our study.

An external threat can also be the posts regarding ML frameworks e.g., CoreML, Azure, etc. However, to limit the scope, we have focused on Machine Learning libraries and did not consider the posts related to the ML framework in our dataset.

3.3 Analysis and Results

We have proposed four research questions to understand what developers ask about ML. We explore the answers to these research questions in this study. The research questions cover the following aspects: identifying the difficult stages in the current ML pipeline faced by the developers (**RQ1**), understanding whether the problems faced by the developers are only due to the design of library or there are some problems inherent to ML (**RQ2**), exploring whether some of the libraries are more difficult in certain stages and are

Table 3.2: Reputation of users for posts in our study.

Library	Min	Max	SD	Mean	Median
Caffe [9]	0	23597	5605	1718	181
H2O [10]	30	168	42	66	50
Keras [11]	0	44553	2392	451	130
Mahout [12]	20	160	34	56	45
MLlib [13]	25	12594	1194	230	65
scikit-learn [14]	0	3820	313	213	100
Tensorflow [15]	0	12265	998	477	170
Theano [16]	20	1135	221	157	66
Torch [17]	0	1234	358	232	55
Weka [18]	25	1603	168	99	49

Table 3.3: Percentage of questions in each top-level category across libraries (in %).

	Caffe	*H2O*	*Keras*	*Mahout*	*MLlib*	*scikit-learn*	*Tensorflow*	*Theano*	*Torch*	*Weka*	Q1	Q3	IQR	Median	SD
Data Preparation	14.0	41.0	16.0	17.0	33.0	26.0	16.0	17.0	23.0	30.0	16.5	29.0	12.5	20.0	8.7
Modeling	32.0	24.0	28.0	44.0	29.0	25.0	27.0	27.0	33.0	20.0	26.5	31.2	4.7	27.0	5.5
Training	24.0	18.0	25.0	8.0	15.0	18.0	21.0	16.0	20.0	12.0	15.0	20.7	5.7	18.0	4.7
Evaluation	1.0	6.0	8.0	4.0	7.0	9.0	9.0	3.0	3.0	10.0	3.5	8.3	4.8	6.0	2.9
Tuning	1.0	6.0	0.0	0.0	2.0	4.0	1.0	1.0	0.0	0.0	0.0	1.5	1.5	1.0	1.9
Prediction	6.0	0.0	10.0	4.0	6.0	7.0	4.0	2.0	2.0	11.0	2.6	6.6	4.0	5.0	3.2
Non-ML	22.0	6.0	13.0	23.0	6.0	11.0	20.0	35.0	20.0	10.0	10.3	21.4	11.1	16.0	8.6

$IQR = Q3 - Q1$: inter-quartile range. SD: standard deviation. Gradient color red to represent increasing percentage.

there libraries that show comparable difficulties in all the stages (**RQ3**), exploring whether the problems faced by the developers changed over time or they stayed consistent (**RQ4**). Next, we answer these questions using a statistical analysis summarized in Tables 3.3 and 3.4 and present our findings.

3.4 RQ1: Difficult stages

If we know the relative difficulty of ML stages for developers, then software engineering R&D and educational efforts can prioritize work on challenging stages. This section explores this question.

3.4.1 Most difficult stage

Finding 1 $\Rightarrow$ Model creation is the most challenging (yet critical) in ML pipeline, especially for libraries supporting distributed ML on clusters like *Mahout* and *MLlib*.

Table 3.4: Percentage of questions in each subcategory across libraries (in %).

	Caffe	*H2O*	*Keras*	*Mahout*	*MLlib*	*scikit-learn*	*Tensorflow*	*Theano*	*Torch*	*Weka*	Q1	Q3	IQR	Median	SD
Data Adaptation	9.84	35.29	7.9	14.58	25.2	8.64	9.22	10.41	22.95	20.51	9.37	22.3	12.93	12.5	8.7
Featuring	0	5.88	1.09	0	4.2	9.34	0.74	0.52	0	4.27	0.13	4.3	4.17	0.92	3.03
Type Mismatch	1.52	0	1.09	0	2.52	2.92	2.02	2.08	0	1.71	0.27	2.07	1.8	1.61	1.02
Shape Mismatch	1.52	0	5.5	0	0	1.86	2.62	2.08	0	0	0	2.03	2.03	0.75	1.7
Data Cleaning	1.52	0	0.55	2.1	2.52	3.62	2.09	1.6	0	3.41	0.79	2.4	1.61	1.82	1.22
Model Creation	26.52	17.64	25.88	43.75	23.52	21.37	23.01	23.43	22.95	21.36	21.77	25.3	3.53	23.22	6.7
Model Selection	0	0	0.55	0	0.84	2.1	0.6	0	0	0	0	0.58	0.58	0	0.64
Model Conversion	3.79	0	0.27	0	0.84	0.33	2.25	2.6	4.91	3.41	0.24	3.21	2.97	1.54	1.71
Model Load/Store	1.5	5.88	1.63	0	5.04	1.75	1.94	1.01	4.91	1.71	1.55	4.2	2.65	1.73	1.88
Error/Exception	0.76	5.88	5.5	4.2	5.88	4.78	5.32	5.72	1.64	2.56	2.96	5.67	2.71	5.1	1.8
Parameter Selection	9.1	5.88	5.5	0	2.52	3.97	3.74	2.6	8.2	5.13	2.89	5.78	2.89	4.5	2.6
Loss Function	6.1	0	4.09	0	0.84	1.4	3.74	2.6	3.3	1.71	0.98	3.6	2.62	2.16	1.86
Optimizer	2.3	0	1.09	2.1	0	0.7	2.77	1.04	3.3	0.85	0.74	2.2	1.46	1.07	1.07
Performance	2.3	5.88	6.27	2.1	5.04	3.27	4.87	3.12	1.64	0.85	2.13	5	2.87	3.2	1.8
Accuracy	3.78	0	2.45	0	0.84	3.62	0.9	1.04	1.64	0.85	0.84	2.3	1.46	0.97	1.3
Eval. Strategy Selection	0.75	0	2.18	2.08	5.04	3.85	5.24	0	1.64	8.54	0.84	4.7	3.86	1.86	2.64
Visualization	0	0	1.63	0	0	2.68	1.65	0	0	2.68	0	1.23	1.23	0	0.95
Output Interpretation	0	5.88	3.82	2.1	1.68	2.21	2.17	2.6	1.64	1.71	1.69	2.5	0.81	2.13	1.47
Tuning Strategy Selection	0.75	5.88	0.27	0	1.68	3.5	0.45	1.04	0	0	0.07	1.52	1.45	0.6	1.82
Tuning Param Selection	0	0	0	0	0	0.81	0.08	0	0	0	0	0	0	0	0.24
Prediction Accuracy	6.1	0	6.81	4.2	5.04	5.25	3.97	2.08	1.64	8.54	2.55	5.85	3.3	4.6	2.44
Model Reuse	0	0	1.37	0	0	0.23	0.22	0	0	1.71	0	0.23	0.23	0	0.6
Robustness	0	0	1.65	0	0.84	1.28	0.3	0	0	0.85	0	0.85	0.85	0.15	0.59
Non-ML API	2.27	0	2.72	4.2	2.52	2.8	4.4	2.08	3.27	1.71	2.13	3.15	1.02	2.63	1.19
Setup	16.67	5.88	9.53	18.75	2.52	5.95	14.5	30.72	16.39	7.69	6.39	16.6	10.21	12.05	7.9
Custom Code	1.52	0	0.81	0	0.84	1.51	1.05	1.56	0	0.85	0.21	1.4	1.19	0.84	0.6
Bug	1.52	0	0	0	0	0.23	0.07	0	0	0	0	0.06	0.06	0	0.45

IQR and SD are defined in Table 3.3. Gradient color **red** to represent increasing percentage.

As Table 3.3 shows, and as expected the model creation is the most difficult, but surprisingly data preparation is the next difficult stage which turns out to be more difficult than the training stage.

Model creation has a median of 23% across all the libraries which is the highest compared to all other stages in the ML pipeline. Some of the libraries for distributed ML e.g., *Mahout*, *Torch*, *Caffe*, *MLlib* have abnormally high difficulty in the model creation stage. This suggests that machine learning in a distributed environment is not developer friendly yet. To understand the reason behind the model creation related problems, we have studied the posts labeled as model creation in *Mahout* and have found that 40% of the model creation related posts are about the recommendation engines. There are different types of models needed to approach different types of recommendation systems, e.g., item-based, user-based recommendation systems. Also, users develop custom recommendation systems for their need and [94] has studied

different recommendation systems and benchmark them to notify developers regarding the better choice of recommendation system based on their intent.

[95] has built a *Keras* based tool that helps the developer to create the ML model from the user's intent. [96] has built a similar tool for *Weka*. Also, finding 1 indicates that, similar tool support in creating models, especially in distributed machine learning is needed.

Further analysis showed that libraries that model creation is especially harder for ML libraries that require developers to use multiple configuration languages to configure their models, for example in *Caffe*. For example in *Caffe*, questions about using multiple languages are discussed frequently. According to a case study by Amershi *et al.* [34], Microsoft developers face similar issues in the machine learning pipeline. It says that though Data Availability, Collection, Cleaning, and Management are most challenging for all three groups of the developer but Model Evolution, Evaluation, and Deployment are more significant for all groups according to the frequency. Our results from studying the posts support what was known as a corporate community and have found several new findings.

We have conducted an expertise analysis of the developers asking questions. We have used the library-specific reputation in *Stack Overflow*. From Table 3.6, we have found that the median reputation of the developers asking questions regarding model creation ranges from 45-270. This indicates that model creation is hard for developers with varied expertise level.

3.4.2 Data preparation

Finding 2 ⇒ Data preparation, especially data adaptation, is the second most difficult stage in ML pipeline.

This top-level category includes questions about adapting the data to the format required by the library, featuring, dealing with type and shape mismatches, and data cleaning. Altogether, this stage is the next most difficult stage across ML libraries (median 20%).

Further analysis showed that ML libraries that use uncommon formats lead to additional difficulties among the developers to understand the format, use the new formats in their software, data wrangling and preprocessing to the format of the data. For example, *Weka*, *MLlib* have a higher problem in data adaption due to their use of uncommon ARFF and RDD formats of data. For some libraries, data preparation turns out to be even more challenging compared to model creation. For example, *H2O*, *Torch* and *Weka* have 35.29%, 22.95% and 20.51% of posts, respectively, about data adaptation.

Finding 2 suggests that the tradeoff in the design of data preparation APIs, e.g. use of custom formats, needs more study. Interestingly, most of the ML textbooks and courses spend little time on data preparation related discussions. Thornton *et al.* [96] has built a tool for *Weka* that not only helps the users develop a model but also performs data wrangling that removes data adaptation-related issues. However, more research is needed that can perform the data wrangling operations for other libraries to support the users' need and matches with the ML models' input.

The library-specific reputation analysis in Table 3.5 shows that these problems are faced by developers with a median reputation in the range of 45-170. So, we have observed that data preparations related questions are asked by the developers with a comparatively less median reputation than the developers asking questions about model creation.

Surprisingly Tuning and Prediction stages of the ML pipeline—topics discussed frequently in the ML research papers—appear infrequently in *Stack Overflow* questions.

3.5 RQ2: Nature of problems

Are some difficulties inherent to ML and thus all ML libraries face them? If so, general solutions could be developed and adapted to all ML libraries. Otherwise, the design of the specific library could be improved by utilizing lessons learned in this section.

3.5.1 Type mismatch

Type mismatch questions have a median of 1.61%, SD of 1.02%, and IQR of 1.80%. The smaller IQR indicates that type mismatch appears in most of the ML libraries. *scikit-learn*, *MLlib*, *Theano* and *Tensorflow* have higher difficulties in type-related problems with 2.92%, 2.52%, 2.08% and 2.02%, respectively. *MLlib* uses a custom data format called RDD that seems to make type-related problems more frequent for this library. There are also questions about failures due to type mismatch in *scikit-learn*, *Tensorflow*, and *Theano* as their APIs have type requirements that are not currently checked.

Finding 3 suggests that ML libraries have not focused on type correctness and ML-specific type correctness. A detailed work in this direction is needed to solve several type-related bugs and failures of the ML models. A static analysis tool might be able to prevent the majority of these problems. To understand the characteristics of the type mismatch related posts, we randomly select 44 *Stack Overflow* posts. We found 31 out of 44 problems were caused by the abstraction created by the libraries to create ML types. The other 13 were standard Python type errors. As an example, the following exception is thrown due to an ML type error.

```
ValueError: ('Unknown loss function', ':root_mean_squared_error')
```

Table 3.6 shows that in *Theano*, these questions are asked by developers with a relatively lower reputation, whereas in *Tensorflow*, developers with a comparatively higher reputation faced these problems.

3.5.2 Shape mismatch

Finding 4 ⇒ Shape mismatch problems appear frequently in deep learning libraries. *Keras* is an outlier in this subcategory with 5.5% of posts.

Shape mismatch related questions have a median of 0.75%, SD of 1.70%, and IQR of 2.03%. This problem appears in all deep learning library in which *Keras* is an outlier with 5.50%. In these libraries, shapes of neurons at adjacent layers must be compatible otherwise the library will throw exceptions during training or fail during prediction. An example is shown in Figure 3.4.

TensorFlow ValueError: Cannot feed value of shape (64, 64, 3)
for Tensor u'Placeholder:0', which has shape '(?, 64, 64, 3)'

Figure 3.4: Question 40430186: An example showing dimension or shape mismatch problem in training in ML.

We have analyzed further and found that in *Keras*, the shape mismatch related problems are caused due to the confusion between input_shape and input_dim parameter, the channel first and channel last, and the shape of the dense layers. In *Keras*, there are two different types of parameters that conform to the size of the input for the input layer, input_shape, and input_dim. The operation of these two parameters is similar but the definition of them is different. Similarly, in *Keras*, we have found that users have confusion among the shape of the input that includes channel first (The dimension of the input to be first parameter e.g., (1,32,32)) and channel last (the dimension of the input is the last parameter e.g., (32,32,1)). Apart from these two problems, we have found that the inner structure of the dense operation has caused problems. The dense operation works on a 1D tensor and if the tensor is not 1D, instead of throwing an error, it executes the operation on the first dimension of the tensor, e.g., if the tensor size is (2,3), then dense will operate assuming the input shape is a 1D tensor of size 2. Finding 4 suggests that techniques for verifying shape and dimension compatibility are needed for deep learning libraries. Such techniques could verify if the data conforms to model architecture, and dynamic modification of the network against data shape.

Abstract APIs that hide the details of the inner-working of the deep learning networks can further complicate matters. To illustrate consider the following *Keras* code.

```
1 def CreateModel(shape):
2   if not shape:
3     raise ValueError('Invalid shape')
4   logging.info('Creating model')
5   model = Sequential()
6   model.add(LSTM(4, input_shape=(31, 3)))
7   model.add(Dense(1))
8   model.compile(loss='mean_squared_error', optimizer='adam')
9   return model
```

The error is at line 6 where an invalid value of $(31, 3)$ is passed to $input\_shape$. The accepted answer suggests that $input\_shape$ should be $(32, 1)$ instead. The user could not verify statically whether the built model has a compatible shape or if there are any unconnected or extra ports while building the model. To alleviate these problems, a tool is needed that can symbolically analyze the ML model and finds the correct parameter to satisfy the operation. [72] implemented the concolic testing (concrete symbolic) [97] to find the testing coverage, Lipschitz's continuity etc. A similar tool can be developed that can proof the satisfiability of an ML model based on the parameter and the structure.

To understand the reason behind the *Keras* being an outliner in the Shape Mismatch subcategory, we have selected 60 random posts from the dataset. We have found that 21 out of 60 shape mismatch problems are from Keras and the shape mismatch in *Keras* occurs due to the abstraction of APIs used to create layers in the network. The dimension of the layers violates the contracts between the layers without giving any hints to the developer.

3.5.3 Data Cleaning

Finding 5 $\Rightarrow$ Most libraries have problems in data cleaning.

As shown in Table 3.4, data cleaning related questions across the libraries have a median of 1.82%, SD of 1.22% and IQR of 1.61%. Most of the libraries have questions about the data cleaning stage except for

H2O and *Torch*. This is not surprising since data cleaning is an integral part of any data science pipelines. Libraries *scikit-learn*, *Weka* and *MLlib* have the most questions. Also, in these libraries developers with relatively lower reputation asks the questions. Developers with higher reputation find data cleaning difficult with *Keras* and *Caffe*.

Finding 5 suggests that tool support for data cleaning is needed, but such techniques may need to overcome inherent technical challenges. The abstract APIs in these libraries sometimes make cleaning fail. For example, the `nan` values in the `dataframe` needs to be converted first into numpy `nan` type before they can be cleaned using APIs provided by *scikit-learn*. Furthermore, these failures do not indicate the root cause making diagnostics difficult.

3.5.4 Model creation

In model creation subcategory, *the most difficult stage according to RQ1*, we see problems that are both inherent to ML, and specific to design choices in the library. The inherent difficulty of distributed ML is a major source of questions, e.g. see Figure 3.5.

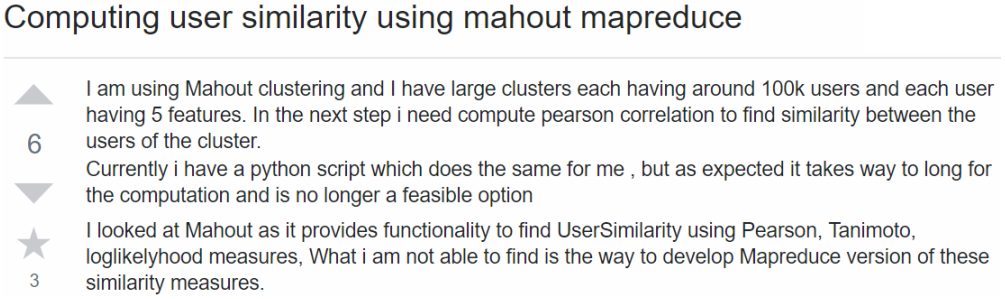


Figure 3.5: Question 12319454: An example question on model creation for distributed ML using *Mahout*.

Deep learning libraries like *Caffe*, *Keras*, *Theano*, and *Tensorflow* also have higher percentages of questions about model creation with 26.52%, 25.88%, 23.43%, and 23.01%, respectively. This shows that model creation for deep neural networks is difficult as well.

When we study the questions about *Caffe* we see that *Caffe* users have problems in model creation due to the dependency of the model on multiple files. To create a model successfully, one needs to make a schema file in protobuf format, create a solver file and write code in C++ or Python to build the model [98]. Having

several components complicates matters. In our study, 36 out of 135 questions about *Caffe* are about model creation problems.

3.5.5 Error/Exception

Finding 6 ⇒ Questions on exceptions/errors are prevalent.

. This indicates static and dynamic analysis tools are needed for such libraries.

Error/Exception subcategory has a median of 5.10%, SD of 1.80% and IQR of 2.71%. All the libraries have issues on runtime error/exception. Surprisingly, though model creation seems problematic in *Caffe*, runtime failure is very low in *Caffe* with 0.76%. *MLlib*, *H2O*, *Keras*, *Tensorflow* and *scikit-learn* have higher percentage of runtime errors with 5.88% and 5.88%, 5.50%, 5.32% and 4.78% respectively.

Finding 6 suggests that debugging and monitoring facilities for ML needs much improvement to help developers resolve error/exception independently. We dug deeper to determine where debugging and monitoring might be most helpful and found that deep learning and distributed ML libraries have more posts about runtime errors at training time, e.g. when a model is throwing an exception at training time, a model is not converging or learning as the iteration of training goes on, a model is not predicting well, etc. Some recent work has started to address these issues [99, 100], but much more work is needed. Due to the lack of debugging tools to monitor pipeline causes of failure are hard to identify. More abstract deep learning libraries throw more runtime exception during training, e.g. see Figure 3.6.

Table 3.5: Library specific reputation in each top-level category across libraries (in median).

	Caffe	*H2O*	*Keras*	*Mahout*	*MLlib*	*scikit-learn*	*Tensorflow*	*Theano*	*Torch*	*Weka*
Data Preparation	85.0	45.0	125.0	47.5	45.0	110.5	170.0	66.5	75.0	55.0
Modeling	270.0	49.0	105.0	55.0	90.0	95.0	155.0	75.0	57.5	45.0
Training	145.0	85.0	145.0	64.0	80.0	95.0	178.0	55.0	73.0	39.0
Evaluation	90.0	80.0	190.0	31.0	629.5	115.0	226.0		98.0	75.0
Tuning	205.0	75.0				100.0	175.0	979.5	69.0	
Prediction	218.5		125.0	30.0	521.0	115.0	85.0	192.5	73.0	60.0
Non-ML	208.0	35.0	135.0	40.0	40.0	82.5	175.0	57.5	52.5	36.0

Blank cell represents the non-availability of the Stack Overflow post in that category in the dataset.
Gradient color red to represent increasing library specific reputation.

Table 3.6: Library specific reputation in each subcategory across libraries (in median).

	Caffe	*H2O*	*Keras*	*Mahout*	*MLlib*	*scikit-learn*	*Tensorflow*	*Theano*	*Torch*	*Weka*
Data Adaptation	85.00	42.50	120.00	45	60.00	120	205.00	119	45	55.00
Featuring		160.00	95.00		115.00	91.5	220.00	40		35.00
Type Mismatch	99.00		80.00		88.00	115	160.00	40		94.00
Shape Mismatch	90.00		164.00			92	90.00	100		
Data Cleaning	326.50		1136.50	50	140.00	210	137.50	48		129.00
Model Creation	247.50	50.00	100.00	55	60.00	95	140.00	75	70	60.00
Model Selection			1015.00		35.00	100	165.00			
Model Conversion	125.00		751.00		25.00		187.50	435.5	40	27.50
Model Load/Store	12378.00	40.00	272.00		60.00	100	213.00	85	521	45.50
Error/Exception	55.00	30.00	120.00	34	45.00	113	150.00	55	930	153.00
Parameter Selection	145.00	168.00	137.50		73.00	107.5	179.00	151	55	32.50
Loss Function	99.00		160.00		55.00	80	190.00	35	617	36.50
Optimizer	35.00		130.00	85		122.5	213.00	42.5	120	40.00
Performance	243.00	85.00	145.00	156	137.50	57.5	189.00	230	105	273.00
Accuracy	182.50		195.00		69.00	80	182.50	27.5	0	35.00
Eval. Strategy Selection	90.00		192.50	42	98.00	140	217.50		25	90.00
Visualization	318.00		319.00			145	318.00			
Output Interpretation		80.00	185.00	20	95.00	90	218.00	35	1234	62.50
Tuning Strategy Selection	195.00	75.00	45.00		69.00	105	195.00	979.5		
Tuning Param Selection	135.00					50	135.00			
Prediction Accuracy	218.50		130.00	30	94.00	105	85.00	332.5	521	52.50
Model Reuse			358.00			85	116.00			69.00
Robustness			103.00		60.00	180	67.50			85.00
Non-ML API	270.00		326.50	45	45.00	102.5	206.50	57	286	30.50
Setup	179.00	35.00	120.00	40	50.00	75	155.00	52.5	39	55.00
Custom Code	519.00		290.00		12594.00	98	312.50	106		25.00
Bug	651.50					0	7000.00			

Blank cell represents the non-availability of the Stack Overflow post in that category in the dataset.
Gradient color red to represent increasing library specific reputation.

3.5.6 Parameter selection

Finding 7 ⇒ Parameter selection can be difficult in all the ML libraries.

We expected parameter selection to be an inherent ML issue but found some variation between libraries, a median of 4.50%, SD of 2.60% and IQR of 2.89%, suggesting key differences among libraries. *Caffe* and

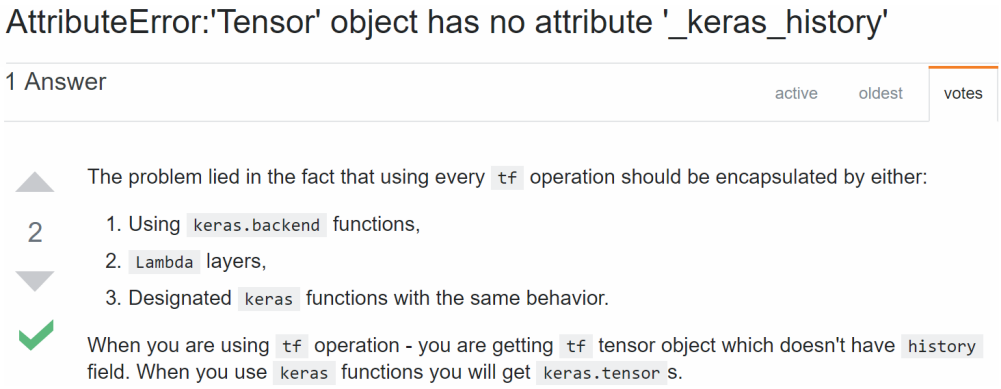


Figure 3.6: Question 45030966: An example question about *Keras* showing abstraction in deep learning libraries could make identifying root cause of an error/exception difficult.

Torch have comparatively more problems with 9.10% and 8.20%, respectively. Libraries like *Keras*, *Weka*, *H2O*, *MLlib* shows larger percentage of questions on choice of parameters.

We have found that in *Caffe* and *Torch*, `batch-size` (25% of the problems) of the data plays a significant role [101] in the learning process. *Caffe* and *Torch* is based on the stochastic gradient descent search optimization, where decreasing the `batch-size` increases the accuracy as well as the training time for an ML model. For selecting parameters adding support for meta-heuristic strategies in the libraries can be helpful.

3.5.7 Loss function selection

Finding 8 ⇒ Choice of the loss function has an impact on the performance and the robustness of an ML model.

Loss functions are used to quantify the difference between values predicted by the model and actual values (labels). Our results show that developers have difficulty selecting an appropriate loss function but the extent of difficulties varies across the libraries (median of 2.16%, SD of 1.86% and IQR of 2.62%). All deep learning libraries have comparatively more questions about loss function, for example, *Caffe*, *Keras*, *Tensorflow* and *Torch* have the highest percentages of 6.10%, 4.09%, 3.74% and 3.30%. Also, these issues are prevalent accross different level of expertise (median reputaion in the range of 35-617) in each library as shown in the Table 3.6. We have studied the posts related to the loss function in *Caffe*, *Keras*, *Torch*, and *Tensorflow* and have found that the primary cause of the problems is the confusion among different

types of built-in loss functions in each library. We have found that *Caffe*, *Keras*, *Tensorflow*, and *Torch* have 8, 14, 17, and 9 built-in loss functions, respectively. These loss functions are problem dependent and it changes if the statement or the intent of the problem changes. Also, 13 out of 46 questions related to the loss functions in *Tensorflow* are related to the gradient calculation for building custom loss functions. A preliminary work [102] has been done on the efficient calculation of the gradient computation. This indicates the necessity of further research on the usage of the loss function in deep learning libraries, e.g. on loss function recommendation. The selection of the loss function is primarily dependent on the type of problem. A wrong selection of the loss function can cause a machine learning model to perform poorer (low accuracy) or can decrease the security of a model by decreasing the robustness that can be utilized by attackers to perform adversarial attack[103].

3.5.8 Training accuracy

Finding 9 ⇒ Abstract ML libraries have higher percentage of questions about training time accuracy and convergence.

We expected training accuracy to be an inherent ML issue impacting all libraries; however, there are few questions about this on *Stack Overflow*. *Caffe* and *scikit-learn* stood out with 3.78% and 3.62% questions about training accuracy. These libraries provide highly abstract APIs and a large number of optional parameters that need to be selected.

This suggests that the library documentation could be clearer about the impact of optional parameters on training accuracy. Secondly, the recommendation system could be developed for parameter recommendations based on dynamic traces.

3.5.9 Tuning parameter selection

Finding 10 ⇒ *scikit-learn* has more difficulty in hyperparameter tuning compared to other libraries

Like accuracy, we considered tuning parameter selection to be an inherent ML issue, impacting those libraries more that have a higher number of parameters. Even though not too many libraries have questions about it, *scikit-learn* and *Tensorflow* stand out. *Tensorflow* has higher usage and questions in general, but *scikit-learn* was as expected due to a large number of optional parameters.

As an example, consider creating `AdaBoostClassifier` with 5 optional parameters initialized to some default values shown below.

```
class sklearn.ensemble.AdaBoostClassifier(
base_estimator=None, n_estimators=50, learning_rate=1.0, algorithm='SAMME.R',
    random_state=None)
```

The `base_estimator` is set to `None` but the user may need to choose an estimator to get the best performance. The learning rate is by default set to 1.0. At this learning rate, it is highly likely that the model will not learn anything. So the user may often use these APIs incorrectly and wonder why the ML model is not producing useful results. Since these are optional parameters, the user will not even get any error or warning. Finding good values for these parameters and tuning them to make the best model, avoiding overfitting are frequent questions among developers using *scikit-learn*. There have been some GitHub issues filed to the repository of *scikit-learn* as bugs (See Figure 3.7 for an example) but the underlying problem was that the developer was not able to trace why the model is not showing expected accuracy, and unable to tune hyperparameters. We have found 36 questions out of 849 in *scikit-learn* asking help about hyperparameter tuning.

In *scikit-learn*, 32 out of 33 questions in parameter selection are related to the gridSearchCV. We have found that there are 11 tunable parameters present for gridSearchCV. The choice of the parameter affects the estimation during training that can be achieved by the grid search process for tuning parameters. There are works [104] that eases the parameter selection for the estimation process. Still, this remains an open research area for further study.

Overall, our results from this and two previous subsections suggest that *parameter recommendation is an urgent need for ML libraries, especially those that have a lot of optional parameters*.

SVC() and SVC(probability=True) give inconsistent predictions #4800

amueller commented on Jun 1, 2015 Owner

You don't tune C and gamma. Try `sklearn.svm.SVC(C=10.0, gamma= 0.01)` and it will work.

Figure 3.7: *scikit-learn* issue #4800: An example of hyperparameter tuning problem. The user filed a bug report, but a developer of the library responded that the problem was with hyperparameter tuning.

3.5.10 Correlation between libraries

Next, we study whether the pattern of problems exhibited by libraries have similarities. The correlation between libraries based on a common pattern of problems is shown in Figure 3.8. We have identified two major groups.

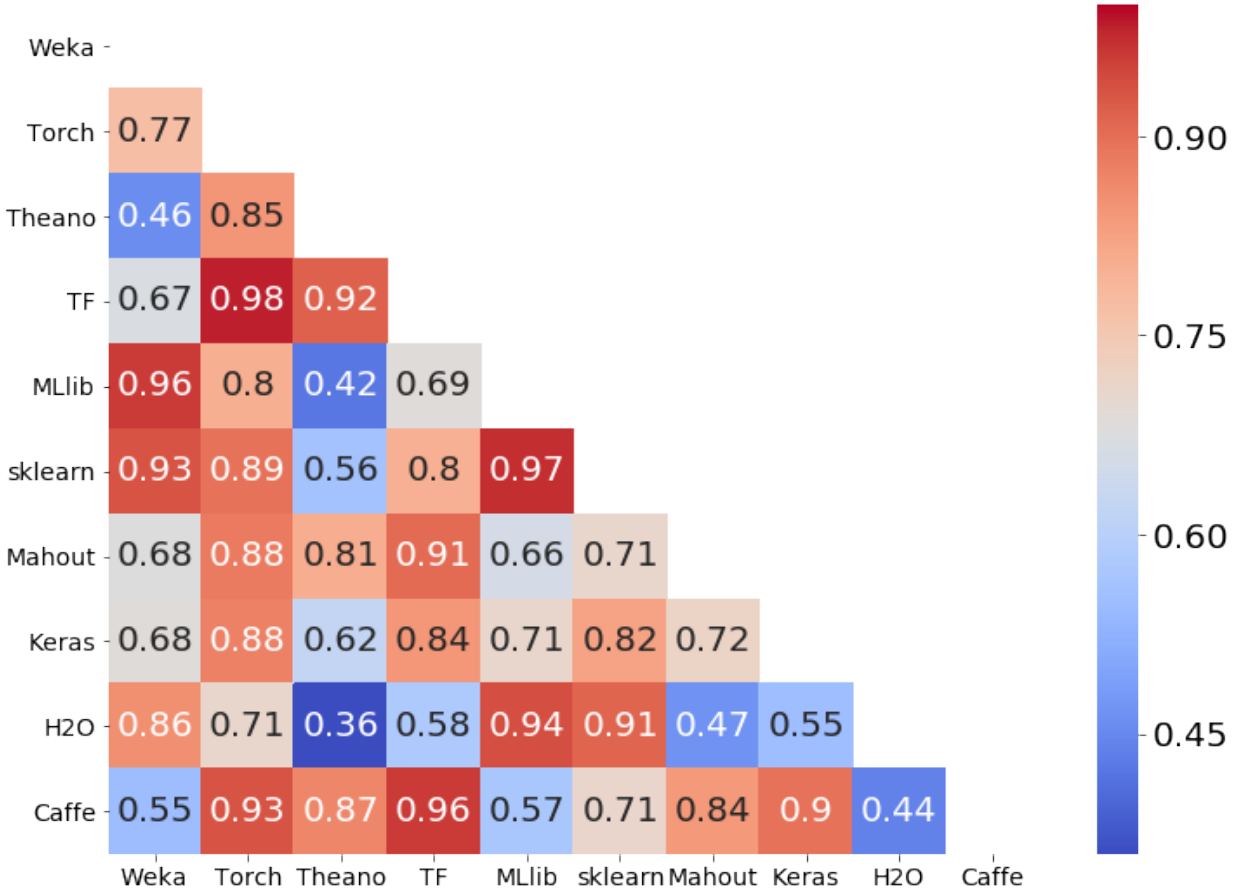


Figure 3.8: Correlation between distributions of percentage of questions over stages of the libraries.

Finding 11 ⇒ *Weka*, *H2O*, *scikit-learn*, *MLlib* form a strongly correlated group with a correlation coefficient greater than 0.84 between the pairs indicating that these libraries have a similar problem in all the ML stages.

Group 1. *Weka*, *H2O*, *scikit-learn*, and *MLlib* form a strongly correlated group with a correlation coefficient greater than 0.84 between the pairs. This suggests that the problems appearing in these libraries have some correlation and the difficulties of one library can be described by the difficulty of other libraries in the group.

This finding is interesting because other than *H2O*, other libraries in this category don't support deep learning. We believe that the correlation may be because each of these libraries supports many different ML algorithms and allows the user to select an algorithm for their tasks. This design choice is markedly different from the other group that is specialized for a single ML algorithm.

Group 2. *Torch*, *Keras*, *Theano*, and *Tensorflow* form another group with a strong correlation of more than 0.86 between the pairs. These libraries are all specialized for deep learning.

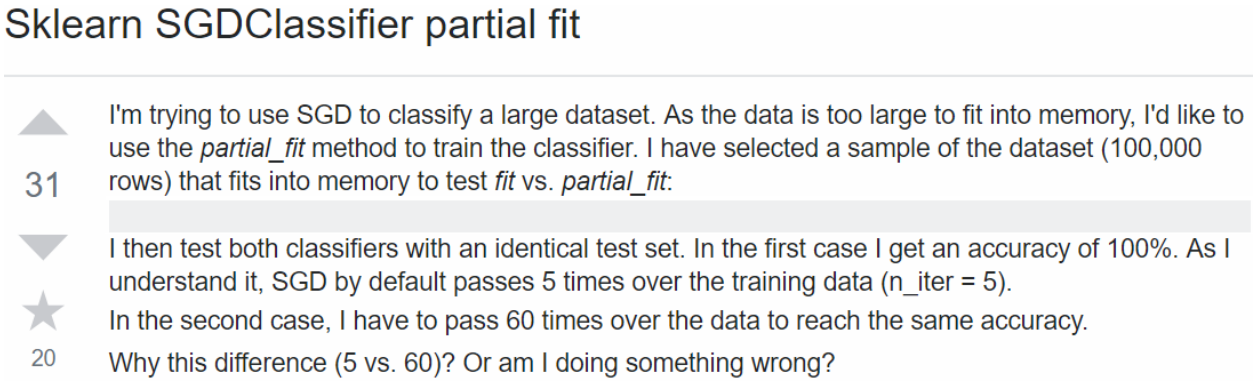
Finding 12 ⇒ Deep learning libraries *Torch*, *Keras*, *Theano* and *Tensorflow* form another group with strong correlation of more than 0.86 between the pairs indicating these libraries follow similar problem in all the stages

his finding is interesting because each of these deep learning libraries has adopted different designs and philosophies. *Tensorflow* and *Torch* are focused on providing low-level general facilities, *Keras* focuses on high-level abstractions, whereas *Theano* focuses on efficiency on both CPU and GPU. Our finding suggests that despite different design philosophies followed by each of these ML libraries, the problems are inter-related for the libraries in this category. So, the software engineering research results for one library may generalize to other deep learning libraries.

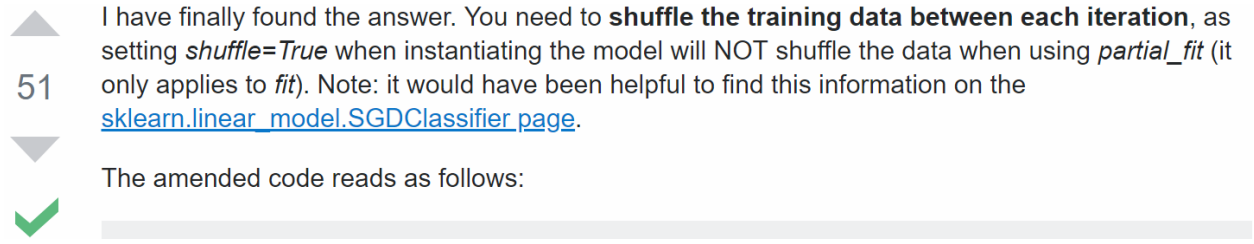
3.5.11 API Misuses in All ML Stages

The ML libraries have APIs that are very often misused. To identify the misuses we have studied both the questions asked by some developers and the well-accepted answers. If the answers pointed out to

incorrect or wrong use of API and provided a solution using the correct use of APIs, we marked them as posts containing API misuse. API misuse is seen across all the stages of ML pipeline.



(a) Question



(b) Best accepted answer

Figure 3.9: Question 24617356: An example showing the API misuse problem in ML libraries. Code snippets are omitted.

For example, see Figure 3.9 where a user is asking that their training takes much time or longer number of iterations to get a certain training accuracy. When they use one API they are able to achieve the desired accuracy in 5 iterations wherein the other API they need 60 iterations to reach the same accuracy. The second API works fine, without any error and eventually reaches the same accuracy. But still, the user is puzzled that almost 12 times higher number of iterations are required when using the second API. The answer in Figure 3.9b suggests that the second API needs the data to be shuffled properly before passing to the API in every iteration. Making that change solves the performance problem. This is an example of API misuse where the precondition of the second API is not satisfied which leads to a performance bottleneck. For another example, let's consider a problem related to the creation of a NaiveBayes model. Only a part of the code snippet where API misuse occurred is shown below:

```
def convert_to_csr_matrix(vectors):
  logger.info("building the csr_sparse matrix representing tf-idf")
  row = [[i] * len(v) for i, v in enumerate(vectors)]
  row = list(chain(*row))
  column = [j for j, _ in chain(*vectors)]
  data = [d for _, d in chain(*vectors)]
  return csr_matrix((data, (row, column)))
```

The code failed to work successfully giving dimension mismatch error in some parts of the code. The solution to the problem is to properly use the API $csr\_matrix()$. This API needs to have a shape parameter defined explicitly and the correct way to use the API is to explicitly define the shape shown in the code below.

```
return csr_matrix((data, (row, column)), shape=(len(vectors), dimension))
```

We have observed another kind of API misuse due to API updates by the library provider. To illustrate, consider the code below that worked well in Apache Spark *MLlib* version < 2.0. For Apache Spark version $>= 2.0$, this API doesn't work. This is one of the top-voted questions on Apache Spark *MLlib* category.

```
1 from pyspark.mllib.clustering import KMeans
2 spark_df = sqlContext.createDataFrame(pandas_df)
3 rdd = spark_df.map(lambda data: Vectors.dense([float(c) for c in data]))
4 mdl = KMeans.train(rdd, 2, maxIterations=10, runs=30, initializationMode="random")
```

MLlib version 2.0 isn't backward compatible and so the code at Line 3 is outdated and must be replaced by the following

```
rdd = spark_df.rdd.map(lambda data: Vectors.dense([float(c) for c in data]))
```

We have found that similar version incompatibility problems are also prevalent in other ML libraries.

Besides, the API misuse scenarios discussed above, many other kinds of API misuse are common in ML libraries, and more detailed analysis and categorization of errors is needed (much like MUBench [105]). Some common problems include failure to find important features, improperly preparing the dataset, perfor-

mance, over-fitting problems, suboptimal prediction performance, etc. A detailed analysis of API misuse is beyond the scope of this work.

3.6 RQ3: Nature of libraries

In this section, we explore whether some of the libraries are more difficult in certain stages and are there libraries that show comparable difficulties in all the stages (**RQ3**). To answer RQ3, we look at three measures. Which libraries have a non-zero percentage of questions under the majority of subcategories? Which libraries have above median percentage of questions under the majority of subcategories? Which libraries have outliers?

It turns out that *scikit-learn* and *Tensorflow* have questions under all subcategories, and *Keras*, *Weka*, *MLlib*, *Caffe*, and *Theano* have questions under the majority of subcategories. On the other hand, *H2O*, *Mahout*, and *Torch* have questions concentrated under few subcategories and other subcategories have no questions. We further observed subcategories under which *H2O* have the majority of questions and found that the majority of the questions are in the initial stages such as how to adapt data to use within *H2O*, how to create a model, or how to setup to use the library adequately. We also observed similar trends for *Mahout* except it has proportionally higher percentage of questions about model creation and setup.

Finding 13 ⇒ Early stages for *H2O* and *Mahout* especially setup and model creation have a comparatively higher percentage of questions compared to later stages.

This may suggest that getting started is harder with *H2O* and *Mahout*. Reflecting further on the nature of *H2O* and *Mahout*, there is a key similarity between the two libraries. Both present non-traditional models of computation to the developers. *H2O* presents a workflow like a model, and *Mahout* is for distributed ML. The absence of questions for later stage subcategories might suggest either that developers who started with *H2O* and *Mahout* stopped using the library or that all developers who faced problems getting started with *H2O* and *Mahout* continued using the library without any major difficulties, and had no questions. Further research is needed to understand which was the case and we didn't find any definitive evidence during this study to suggest either way.

Next, we look at libraries that have an above-median percentage of questions under the majority of subcategories. At the top, >50% subcategories, are *Tensorflow* (20 subcategories), *scikit-learn* (19 subcategories), *Keras* (17 subcategories). *Tensorflow* and *Keras* are popular libraries for deep learning, and above average interest in the majority of the aspects of their functionality reflects their popularity. *scikit-learn* is a popular ML library in Python. Though it is not used for deep learning, its use for regression, supervised and unsupervised learning, and recommendation related tasks are well known. This library provides abstract APIs that hide the details of ML. In our study, the majority of questions about *scikit-learn* were about data preparation (26%), modeling (25%), and training (18%). In the middle, >30% subcategories, we have *MLlib* (13 subcategories), *Caffe* (12 subcategories), *Weka* (11 subcategories), *Theano* (11 subcategories), and *Torch* (9 subcategories). At the bottom, we have *Mahout* (6 subcategories) and *H2O* (8 subcategories). We have previously observed that *Mahout* and *H2O* have questions observed under a few categories associated with initial stages. Combining with this observation suggests that such difficulties are higher for *Mahout* and *H2O* compared to other libraries.

Next, we look at outliers. For shape mismatch *Keras* is an outlier, for model creation, *Mahout* is an outlier, for model selection, *scikit-learn* is an outlier, for output interpretation *H2O* and *Keras* are outliers, for tuning strategy *H2O* is an outlier, for tuning parameter selection *scikit-learn* is an outlier, for model reuse *Keras* and *Weka* are outliers, and for bug *Caffe* and *scikit-learn* are outliers.

Finding 14 ⇒ *scikit-learn* is an outlier in several categories suggesting that a deeper look into its API design might be necessary to improve the usability of this important library.

scikit-learn provides a lot of optional parameters to be selected in their APIs, whose values are hard to select yet affect accuracy. That could be the reason why its users have more difficulties in selecting parameters. *scikit-learn* also has an abnormally high percentage of questions about model selection, which is surprising because it is one of the few libraries to provide abstract model selection APIs, but the use of these APIs could be simplified. This calls for research on designing better APIs for *scikit-learn*.

Next, we will look at the error/exception related questions.

Finding 15 ⇒ Deep learning libraries *Caffe*, *H2O*, *Keras*, *Tensorflow*, *Theano*, *Torch* show more training time difficulties compared to other ML libraries

While this finding shouldn't be a surprise, it reinforces a well-established worry in both the AI/ML and SE/PL communities that explaining why a deep learning model has worked or failed at training time or gives unexpectedly low performance remains a hard and open question. We confirm that it is important to solve it to help developers make effective use of deep learning APIs. To ensure that the dataset represents the usage of these libraries in the open source projects, we have calculated the number of occurrences of these libraries in Github open source projects. Table 3.7 reports the number of occurrences of each library on GitHub. Furthermore, we performed the Kolmogorov Smirnov[106] test among the distribution of the library usage population and our dataset population. We have found p-value of 0.675 and $KS - statistics$ value as 0.3, which suggest that both samples have been taken from a similar population.

Table 3.7: Number of occurrences utilizing the libraries in *Github*.

Library	Occurrences	Library	Occurrences
Caffe	1,46,121	*scikit-learn*	2,69,672
H2O	33,112	*Tensorflow*	39,41,629
Keras	7,55,427	*Theano*	2,28,960
Mahout	2,793	*Torch*	1,21,583
MLlib	90,042	*Weka*	21,779
Overall		56,11,118	

3.7 RQ4: Time consistency of difficulty

In this section, we explore the answer to **RQ4** to understand whether the problems across different stages stayed consistent over time or are there problems that were prominent only for a certain period and then solved by the library developers. To study this question, we plot the percentage of posts across different stages of all the libraries from the year 2009 to March 2018.

Our major observations from Figure 3.10 are described below: **Model creation related problems are consistent over time.** Choice of model problems seems consistent over time indicating model creation

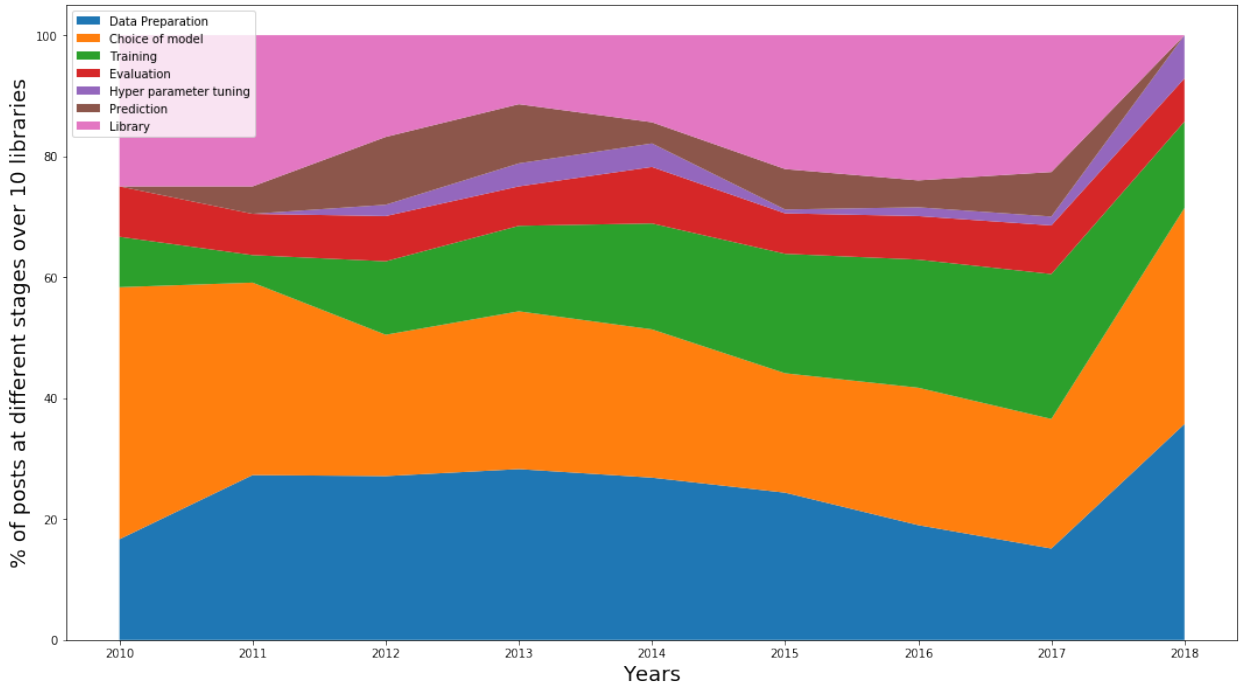


Figure 3.10: Difficulties over time, across different stages

problems are not being affected by the evolution of libraries. While these are fundamental problems for ML, the deeper involvement of SE engineering researchers is needed to glean and disseminate lessons, patterns, and anti-patterns to help ML practice.

Data preparation related problems slowly decrease after 2013 and show a sharp increase after 2017. Weka the library that has most difficulties in data preparation stage started losing popularity and new tensor representation of data gained popularity which explains the slow decline in the data preparation difficulty. The increase in data preparation since 2017 coincides with the increasing interest in deep learning, and popularity of deep learning libraries that provide higher levels of abstraction. Data from a varied set of sources are prepared for deep learning tasks.

Training related problems show a slow increase over time. Due to the popularity of deep learning where training time errors occur more frequently the training related problems are slowly increasing.

Evaluation problems are consistent over time. Evaluation related problems have not been solved by the evolution of ML libraries over the last decade.

3.8 Implications and Discussion

We have studied four research questions in the previous section where we have shown our key findings and analyses. In this section, we shall discuss the implications of the results and the call of action for the software engineering community.

3.8.1 Implications of RQ1

Our analyses and findings on RQ1 provide the following implications:

1. We have seen some of the stages can work as the bottleneck in ML pipeline. So, we need computational techniques to measure the trustworthiness of pipeline, uncertainty that the pipeline can lead to. We also have to think of the design patterns needed for designing the ML pipeline. Currently, when a certain stage fails, we have to restart the process again as the stages are sequential. To find out how can we make the stages more loosely coupled to enhance the production and development needs software engineering innovations. Software engineers need to think about how can we bring the idea of reactive programming, modular programming to the ML pipeline design. Both theoretical and applied techniques are needed to remove the barrier to entry into ML.

2. We need to step back and think whether the current principle of API design is suitable for ML or not. Using traditional abstraction techniques to design ML APIs has found to have limitations in ML. Abstracting away the whole ML computation magic inside an API is not found effective in enhancing productivity. Rather it has shown clear indications of technical debts. So it is a new challenge for the software engineering researcher to devise new principles for designing ML APIs.

3. We have found data preparation works as a barrier to entry in ML. Software engineering researchers can apply the ideas from symbolic analysis, data flow analyses to provide an automatic solution to these problems faced by the ML developers.

3.8.2 Implications of RQ2

In RQ2, we studied the nature of the problems faced by the ML developers. Here we mainly studied whether the root cause of the challenges are from the drawback of libraries or are there new ML specific causes. We have found some ML specific causes like shape mismatch, data cleaning, model creation, parameter selection, loss function selection often creates a barrier to entry. The implications and call of actions for the software engineering researchers from this research question are the following:

1. Shape mismatch problem often causes production failures at the runtime in deep neural networks. We need to adapt the ideas of API contract, temporal analyses between the layers of DNN to solve this kind of error. We might need to think of how can we adapt program analysis techniques to automatically solve this problem.

2. Automatic techniques need to be developed to provide support for model creation. In this endeavor, the long-existing programming by example approaches can be utilized to help ML developers in choosing and creating the right model. The meta-learning techniques can also be improved to help the developers choosing the best performing model given a dataset and a problem.

3. We have identified the challenges faced by the developers in choosing the right hyperparameters, loss function. Currently, the developers use an exhaustive grid search approach to find the correct combination of hyperparameters where the search space is exponential. We need to think about how can we improve this grid search technique using pruning methodologies. We might have to think about how can we take help from mining large scale code repositories to provide a cost-efficient and fast solution to the problem of finding the optimal hyperparameters.

4. ML specific static analysis and dynamic analysis tools, debuggers need to be developed. Researchers and practitioners need to collaborate in developing the theories and tools to solve the existing technical debts in ML.

3.8.3 Implications of RQ3

In RQ3, we have studied the nature of the libraries in terms of the challenges faced by the developers. We have found that libraries doing similar tasks have similar patterns of challenges faced by the developers. For example, the libraries used for doing deep learning have a strong correlation among them in terms of the distribution of the challenges faced by the developers. We derive the following implications from the analyses under this research question:

We have shown the possibility of using the same program analysis tools, bug repair tools in the libraries forming a group in terms of the distribution of the challenges faced by the developers. To achieve this, we might need to convert the program in the source library into a common intermediate representation (IR). Then, we shall do all the analyses and repairing in the IR and finally convert back the fixed program to the model in the original library. We can also convert the programs in one source library to another target library using this common IR. This will help to make the community more connected and we take the better piece of all the libraries to make ML more successful and robust.

3.8.4 Implications of RQ4

In RQ4, we have seen some of the problems faced by the developers like problems in model creation, data preparation, training, evaluation stayed consistent over time. This gives us a negative picture that the unorganized efforts that went so far did not notice any positive outcome. So it has become an immediate need to think about how can we resolve these challenges as a wide talk is going on regarding the usage of ML in safety-critical systems. If we want to trust and allow the usage of ML in these life-threatening safety-critical systems, these problems need to be resolved immediately. And this needs joint effort from the software engineering and machine learning community to develop the necessary theories and tools.

3.9 Conclusion

This study is motivated by the need to empirically understand the problems with the usage of ML libraries. To understand the problems, we retrieved a significant dataset of Q&A from *Stack Overflow* classified these questions into categories and subcategories and performed analysis from four viewpoints:

finding the most difficult ML stage, understanding the nature of problems, nature of libraries and studying whether the difficulties stayed consistent over time. We found that model creation is the most difficult stage followed by data preparation. We found that type mismatch, data cleaning and parameter selection are difficult across all libraries. We also found that initial stages are harder for *H2O* and *Mahout*, and *scikit-learn* has proportionately higher problems in several subcategories. Lastly, we observed that data preparation and training related problems are showing a sign of increase going forward. These findings are a call to action for SE researchers as the engineering of software with ML components is likely to be routine in the next decade. In the future, this dataset can be utilized to train ML models that can classify the new *Stack Overflow* posts. In our current work, we have not studied full-stack ML frameworks e.g., TFX, Azure, CoreML, SageMaker, etc. A detailed study of these frameworks remains an interesting future work.

CHAPTER 4. A COMPREHENSIVE STUDY ON DEEP LEARNING BUG CHARACTERISTICS

4.1 Introduction

A class of machine learning algorithms known as *deep learning* has received much attention in both academia and industry. These algorithms use multiple layers of transformation functions to convert input to output, each layer learning successively higher-level of abstractions in the data. The availability of large datasets has made it feasible to train (adjust the weights of) these multiple layers. While the jury is still out on the impact of deep learning on overall understanding of software's behavior, a significant uptick in its usage and applications in wide ranging areas combine to warrant research on software engineering practices in the presence of deep learning. This work focuses on the characteristics of bugs in software that makes use of deep learning libraries.

Previous work on this topic generally fall under two categories: those that have studied bugs in the implementation of machine learning libraries themselves, and those that have studied bugs in the usage of a specific deep learning library. A key work in the first category is Thung *et al.* [52] who studied bugs in the implementation of three machine learning systems Mahout, Lucene, and OpenNLP. In the second category, Zhang *et al.* [51] have studied bugs in software that make use of the *Tensorflow* library. While both categories of approaches have advanced our knowledge of ML systems, we do not yet have a comprehensive understanding of bugs encountered by the class of deep learning libraries.

This work presents a comprehensive study of bugs in the usage of deep learning libraries. We have selected top five popular deep learning libraries *Caffe* [9], *Keras* [11], *Tensorflow* [15], *Theano* [107], and *Torch* [17] based on the user counts from developers Q&A forum *Stack Overflow*. While each of these libraries are for deep learning they have different design goals. For example, *Tensorflow* focuses on providing low-level, highly configurable facilities whereas *Keras* aims to provide high-level abstractions hiding the low-level details. *Theano* and *Torch* are focused on easing the use of GPU computing to make deep learning

more scalable. Thus, studying them simultaneously allows us to compare and contrast their design goals *vis-à-vis* bugs in their usage.

We have used two sources of data in our study: posts about these libraries on *Stack Overflow* and also *Github* bug fix commits. The first dataset gives us insights into bugs that developers encounter when building software with deep learning libraries. A number of these bugs would, hopefully, be fixed based on the discussion in Q&A forum. The second dataset gives us insights into bugs that were found and fixed in open source software. Our study focuses on following research questions and compares our findings across the five subject libraries.

RQ1: (Bug Type) What type of bugs are more frequent?

RQ2: (Root cause) What are the root causes of bugs?

RQ3: (Bug Impact) What are the frequent impacts of bugs?

RQ4: (Bug prone stages) Which deep learning pipeline stages are more vulnerable to bugs?

RQ5: (Commonality) Do the bugs follow a common pattern?

RQ6: (Bug evolution) How did the bug pattern change over time?

Findings-at-a-glance. Our study show that most of the deep learning bugs are *Data Bugs* and *Logic Bugs* [108], the primary root causes that cause the bugs are Structural Inefficiency (SI) and Incorrect Model Parameter (IPS) [51], most of the bugs happen in the Data Preparation stage of the deep learning pipeline. Our study also confirms some of the findings of *Tensorflow* conducted by Zhang *et al.* [51]. We have also studied some antipatterns in the bugs to find whether there is any commonality in the code patterns that results in bugs. Our findings show that there is strong correlation among the distribution of bugs as well as in the antipatterns.

4.2 Methodology

4.2.1 Data Collection

We have used two different data sources for studying the bugs in deep learning software: *Stack Overflow* posts and *Github* bug fix commits. A summary of these datasets is shown in Table 5.1.

Table 4.1: Summary of the dataset used in the Study

Library	*Stack Overflow*		*Github*	
	# Posts	# Bugs	# Commits	# Bugs
Caffe	183	35	100	26
Keras	567	162	100	348
Tensorflow	1558	166	100	100
Theano	231	27	100	35
Torch	177	25	100	46
Total	2716	415	500	555

4.2.1.1 *Stack Overflow* Data Collection

To study bugs in deep learning software, we have collected data from *Stack Overflow*, a well-known Q&A site for developers to discuss software development problems. The data collection process consists of two steps.

In the first step, we select candidate posts discussing deep learning libraries. We focus on five deep learning libraries: *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch*. These are the five most discussed deep learning libraries on *Stack Overflow*. We did that by searching for posts tagged with *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch*. When posts are about specific libraries, they are more likely to talk about bugs in using deep learning libraries. Using these criteria, we selected all posts about these five libraries. We further filtered the posts that did not contain any source code because posts about bugs usually contain code snippets. Moreover, we reduced the number of posts by selecting the posts whose scores, computed as the difference between the number of its upvotes and the number of its downvotes, were greater than 5 to focus on the high-quality posts and keep the manual effort manageable. After this step, in total, we retrieved 183, 567, 1558, 231, and 177 posts for *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch*, respectively for further study.

In the second step, we manually read these candidates to identify the ones about bugs. After that, the second and the third authors manually reviewed the candidates. For each post, we read the question and all answers focusing on the best-accepted one. If the best-accepted answer was to fix the usages of the deep

learning API(s) in the question, we considered that post as talking about deep learning bugs. After this step, we found 35, 162, 166, 27, and 25 bugs for *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch* respectively.

4.2.1.2 *Github* Data Collection

We mine the *Github* commits to study the change in the commits and to check and confirm the bug patterns that we studied from *Stack Overflow*. The data collection process consists of two steps.

First, we collect all the repositories of *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch*. For collecting the repositories that use these libraries, we first find the repositories that contain the keywords related to the libraries. After that, we mine all the commits whose title contains the word `"fix"`. Then, we check the import statements in the program to identify if those repositories truely use deep learning libraries. Next, we randomly select 100 commits for each library from mined commits and classify them.

Secondly, we use the same process that we used for *Stack Overflow*. Specifically, the second and the third authors manually studied the 500 commits and separately label them. After that, these two authors compare their results to fix the conflict in the labeling process. We study each line of change in the commits. Note that some commits may have more than one bugs and some commit may not have bug. Overall, we got 26, 348, 100, 35, and, 46 bugs for the commits of *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch*, respectively.

4.2.2 Classification

In our classification, we focus on three criteria which are bug types, root causes and effects of bug. The classification scheme used for labeling of the bugs in each of these three criteria discussed in Section 4.2.4, Section 4.2.5, and Section 4.2.6. We have also classified the bugs into different deep learning stages [1].

To label the bug types we followed the classification from an already existing well vetted taxonomy [108] and appended on top of that. The added types were based on the data that we studied following an open coding scheme.

The bugs may have different root causes and effects. A supervised pilot study and open coding schemes were used to identify the effects that are possible through these bugs. We have adapted the classification scheme of root causes and bug effects from [51] and added on top of that as found from the study of the

posts. One of the authors with expertise in these libraries studied the posts initially to come up with the classification scheme for bug types, root causes and effects. We followed the open coding scheme and a pilot study was conducted to get agreement on the classification.

We also classified the bugs into different stages of the pipeline to understand which stages are more vulnerable to bugs. Deep learning process can be divided into seven stage pipeline [1]. The stages are data collection, data preparation, choice of model, training, evaluation, hyper parameter tuning and prediction. Among the seven stages, the first one is not related to software development. The other stages are related to software development, and are supported by the deep learning libraries through their APIs. We use these stages to label the bugs into different stages.

4.2.3 Labeling the Bugs

Once we have all the classification criteria, we used those criteria to label the posts. The second and the third authors independently studied the posts. We measured the inter rater aggrement among the labellers using Cohen's Kappa coefficient [109] when 5%, 10%, 20% , 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100% of the posts were labeled. After 5% labeling, the Cohen's Kappa coefficient was close to 0. Then we conducted a training session among the raters to clarify the labeling and what they mean. After the training session, we conducted another pilot study at 10% including the first 5%. This time the Cohen's Kappa coefficient was 82%. We again discussed the results and find out the reasons for major disagreements. We then discussed those cases further through examples and continued labeling. The Cohen's Kappa coefficient was more than 90% in subsequent pilot studies.

The labeling effort was continuously being monitored with the help of Kappa coefficient to understand the agreement. We conducted reconciling efforts ideally at every 10% interval of the labeling. The posts where there was disagreement between the raters were further discussed in the presence of a supervisor. After discussion and arguments a common label was given. Finally, all the bugs were given a common label.

4.2.4 Types of Bugs in Deep Learning Software

Developers often encounter different types of bugs while trying to write deep learning software. To understand those bugs and their root causes, we have classified them into different categories. The classification is inspired from [108] and adapted based on all the *Stack Overflow* posts that we have analyzed.

4.2.4.1 API Bug

This group of bugs is caused by deep learning APIs. Generally, when a developer uses a deep learning API, different bugs associated with that API are inherited automatically without the knowledge of the user. The prime causes for triggering of deep learning API bugs can be because of the change of API definition with different versions, lack of inter-API compatibility and sometimes wrong or confusing documentation.

4.2.4.2 Coding Bug

These kind of bugs originate due to programming mistakes. This in turn, introduces other types of bugs in the software which lead to either runtime error or incorrect results. A big percentage of the deep learning bugs that we have checked arises from syntactic mistakes that cannot be fixed by changing only some lines of code. This type of bugs are not identified by the programming language compiler resulting in wrong output.

4.2.4.3 Data Bug

This bug may arise if an input to the deep learning software is not properly formatted or cleaned well before supplying it to the deep learning model. This type of bug occurs before data is fed to the deep learning model. It is not because of the wrong deep learning model, rather it is purely based on the type and structure of training or test data. Similar to coding bugs, data bugs are usually flagged by the compiler, but in some scenarios it can pass unchecked through the compilation process and generate erroneous results.

4.2.4.4 Structural Bug (SB)

A vast majority of the deep learning bugs are occurring due to incorrect definitions of the deep learning model's structure. These include mismatch of dimensions between different layers of deep learning models, the presence of anomaly between the training and test datasets, use of incorrect data structures in implementing a particular function, etc. These type of bugs can be further classified into four subcategories.

Control and Sequence Bug This subclass of the bug is caused by the wrong structure of control flow. In many scenarios, due to wrong if-else or loop guarding condition, the model does not perform as expected. This type of bug either leads to a crash when a part of deep learning model does not work or, leads to incorrect functionality due to mishandling of data through the layers.

Data Flow Bug The main difference between the Data Flow Bug and the Data Bug is the place of origin. If a bug occurs due to the type or shape mismatch of input data after it has been fed to the deep learning model, we label it as Data Flow Bug. It includes those scenarios where model layers are not consistent because of different data shape used in consecutive layers. To fix these bugs, developers need to modify the model or reshape the data.

Initialization Bug In deep learning, Initialization Bug means the parameters or the functions are not initialized properly before they are used. This type of bugs would not necessarily produce runtime error but it will simply make the model perform worse. Here, the definition of functions includes both user-defined and API defined. We also categorize a bug into this category when the API has not been initialized properly.

Logic Bug In deep learning, the logical understanding of each stage of the pipeline is an integral part of the coding process. With an incorrect logical structure of the deep learning model, the output of a program may result in either a runtime error or a faulty outcome. These bugs are often generated in the absence of proper guarding conditions in the code.

Processing Bug One of the most important decisions in the deep learning model structure is to choose the correct algorithm for the learning process. In fact, different deep learning algorithms can lead to different

performance and output [110]. Also, to make different layers be compatible with each other, the data types of each layer need to follow a contract between them. Processing Bugs happen due to the violation of these contracts.

4.2.4.5 Non Model Structural Bug (NMSB)

Unlike SB, NMSB occur outside the modeling stage. In other words, this bug can happen in any deep learning stage except the modeling stage such as the training stage or the prediction stage. NMSB has similar subcategories as SB. The subcategories of NMSB are Control and Sequence Bug, Logic Bug, Processing Bug, and Initialization Bug. We do not define Non Model Structural Data Flow Bug like Structural Data Flow Bug because Data Bug already covers the meaning of Non Model Structural Data Flow Bug.

Control and Sequence Bug This subclass is similar to Control and Sequence Bug in SB. The bug is caused by an incorrect structure of control flow like wrong if-else condition; however, this kind of bug happens outside modeling stage.

Initialization Bug This subclass is similar to Initialization Bug in SB. The bug is caused by incorrect initialization of a parameter or a function prior to its use.

Logic Bug This subclass is similar to Logic Bug in SB. The bug is caused by misunderstanding the behavior of case statements and logical operators.

Processing Bug This subclass is similar to Processing Bug in SB. The bug is caused by an incorrect choice of algorithm.

4.2.5 Classification of Root Causes of Bugs

4.2.5.1 Absence of Inter API Compatibility.

The main reason for these bugs is the inconsistency of the combination of two different kinds of libraries. For example, a user cannot directly use `Numpy` function in *Keras* because neither *Tensorflow* backend nor *Theano* backend of *Keras* has the implementation of `Numpy` functions.

4.2.5.2 Absence of Type Checking.

This kind of bugs involves a type mismatch problem when calling API methods. These bugs are usually mistakes related to the use of wrong type of parameters in an API.

4.2.5.3 API Change.

The reason for these bugs is the release of the new versions of deep learning libraries with incompatible APIs. In other words, the bug happens when the new API version is not backward compatible with its previous version. For example, a user updates the new version of a deep learning library which has new API syntax; however, the user does not modify his/her code to fit with the new version, which leads to the API change bug.

4.2.5.4 API Misuse.

This kind of bugs often arises when users use a deep learning API without fully understanding. Missing conditions can be one kind of API misuse, and this bug occurs when a usage does not follow the API usage constraints to ensure certain required conditions. Crash is the main effect of these bugs.

4.2.5.5 Confusion with Computation Model.

These bugs happen when a user gets confused about the function of deep learning API, which leads to the misuse of the computation model assumed by the deep learning library. For instance, a user gets confused between the graph construction and the evaluation phase.

4.2.5.6 Incorrect Model Parameter or Structure (IPS)

IPS causes problems with constructing the deep learning model, e.g. incorrect model structures or using inappropriate parameters. IPS is a common bug in the deep learning software because of both the lack of deep learning knowledge among the users and the incomprehensibilty of deep learning models. This kind of bugs causes the functional incorrectness; thus, the effect of this bug is a crash.

4.2.5.7 Others.

These bugs are not related to deep learninng software. In other words, these bugs are mostly related to mistakes in the development process like incorrect syntax.

4.2.5.8 Structure Inefficiency (SI)

SI causes problems related to modeling stage in deep learning software like IPS; however, SI leads to bad performance of the deep learning software while IPS leads to a crash.

4.2.5.9 Unaligned Tensor (UT)

These bugs often occur in the computation graph construction phase. When a user builds the computation graph in deep learning process, they have to provide correct input data that satisfies input specifications of the deep learning API; however, many users do not know the API specifications, or they misunderstand API signature leading to UT bugs.

4.2.5.10 Wrong Documentation.

Incorrect information in library documentation leads to these bugs. Deep learning library users may face this kind of bugs when they read an incorrect definition or an incorrect usage of a deep learning API from documentation.

4.2.6 Classification of Effects of Bugs

4.2.6.1 Bad Performance.

Bad performance or poor performance is one of common kind of effect in deep learning software. Furthermore, the major root causes of this effect are SI or CCM that are related to the model construction. Even though developers can use deep learning libraries correctly, they still face model construction problems because APIs in these libraries are abstract.

4.2.6.2 Crash.

Crash is the most frequent effect in deep learning. In fact, any kind of bugs can lead to Crash. A symptom of crash is that the software stops running and prints out an error message.

4.2.6.3 Data Corruption.

This bug happens when the data is corrupted as it flows through the network. This effect is a consequence of misunderstanding the deep learning algorithms or APIs. When Data Corruption occurs, a user will receive unexpected outputs.

4.2.6.4 Hang.

Hang effect is caused when a deep learning software ceases to respond to inputs. Either slow hardware or inappropriate deep learning algorithm can lead to Hang. A symptom of Hang is that the software runs for a long period of time without providing the desired output.

4.2.6.5 Incorrect Functionality

This effect occurs when the software behaves in an unexpeced way without any runtime or compile-time error/warning. This includes the incorrect output format, model layers not working desirably, etc.

4.2.6.6 Memory Out of Bound

Deep learning software often halts due to unavailability of the memory resources. This can be caused by, either the wrong model structure or, not having enough computing resources to train a particular model.

4.3 Frequent bug types

In this section, we explore the answer to **RQ1** through a statistical analysis of the labeled data. The normalized distribution of bug types in *Stack Overflow* data is shown in Figure 4.1. The distribution of bugs shown in Figure 4.1 and the *Stack Overflow* and *Github* data in Table 5.3 shows the presence of different kinds of bugs in both *Stack Overflow* and *Github* for the deep learning libraries we have studied. We present some of the key findings related to bug types in the following subsections.

4.3.1 Data Bugs

Finding 1 ⇒ Data Bugs appear more than 26% of the times

From Figure 4.1 we see that among the bug types the Data Bugs frequently appear (26%) in all the libraries. In the studied *Stack Overflow* data, we have seen 30% of the posts in *Tensorflow*, 24% posts in *Keras*, 36% posts in *Torch*, 35% posts in *Theano*, and 9% posts in *Caffe* have Data Bugs. Data bugs mostly appear due to the absence of data pre-processing facilities like feature engineering, data validation, data shuffling, etc. For example, a developer is trying to read some image files using the following method[1].

```
def _read32(bytestream):
  dt = numpy.dtype(numpy.uint32).newbyteorder('>')
  return numpy.frombuffer(bytestream.read(4), dtype=dt)
```

The developer eventually got stuck with the following error while trying to train the model using the data returned by the previous library call.

```
TypeError: only integer scalar arrays can be converted to a scalar index
```

[1] https://tinyurl.com/y3v9o7pu

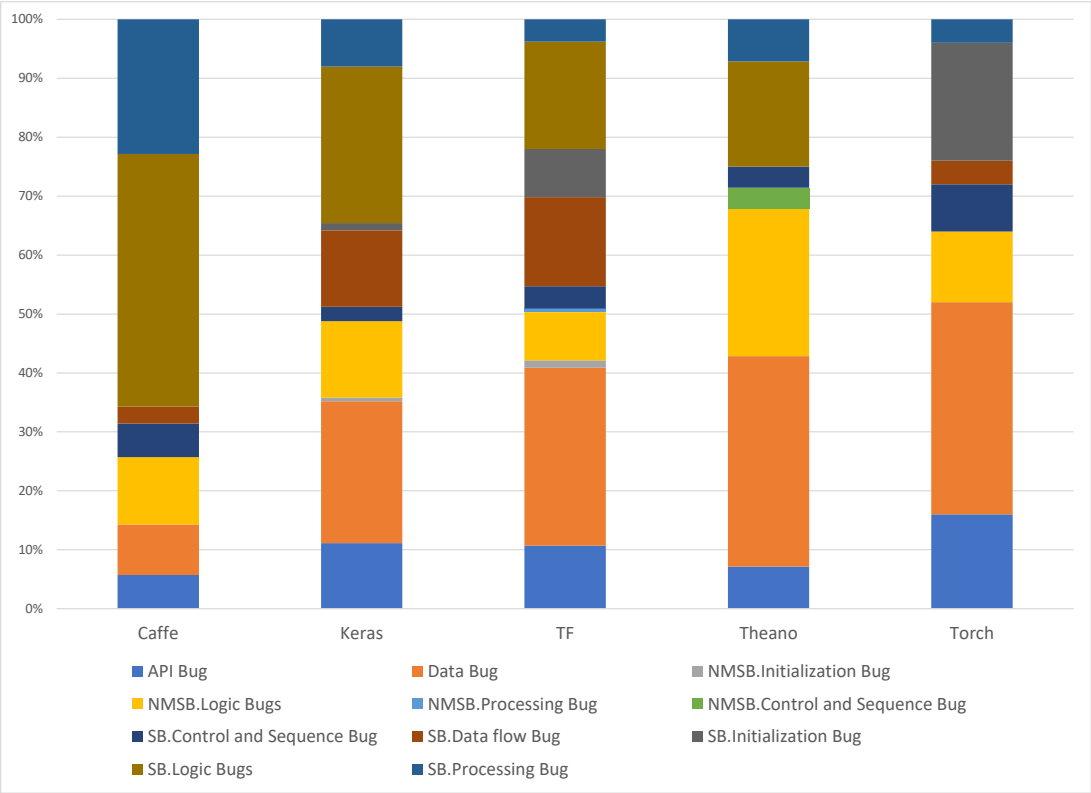


Figure 4.1: Distribution of Bug Types in *Stack Overflow*

An expert suggested an answer to change the last return statement with the following, which solved the problem and was accepted.

```
return numpy.frombuffer(bytestream.read(4), dtype=dt)[0]
```

The bug is hard to fix by just looking at the error message. It is difficult to identify the exact reason of bug which led the developer to post a question on *Stack Overflow* and the question was upvoted by other fellow developers as a qualified post.

The large percentage of Data Bugs indicate data pre-processing related difficulties are quite common in deep learning software. These bugs could be addressed by development and refinement of data verification

tools. Support for modern abstract data types like *DataFrame* and the properties of the model in data verification tool would help the deep learning community.

4.3.2 Structural Logic Bugs

Finding 2 ⇒ *Caffe* has 43% Structural Logic Bugs

The second major bug type is Structural Logic Bug in *Stack Overflow* that was expected from our initial hypothesis based on a pilot study. *Caffe* has more Structural Logic Bugs in *Stack Overflow* compared to other libraries. Other libraries also have significant portion of Structural Logic Bugs ranging from 0% - 27%.

4.3.3 API Bugs

Finding 3 ⇒ *Torch*, *Keras*, *Tensorflow* have 16%, 11% and 11% API bugs respectively

In deep learning libraries API changes sometimes break the entire production code. The implicit dependencies between libraries cause problems when one library has some major changes. For example, when `Numpy` is updated *Tensorflow*, *Keras* software may fail. *Keras* often uses *Tensorflow* or *Theano* as backend and hence update of *Tensorflow* or *Theano* can cause the software developed using *Keras* to crash. API bugs arise more often in *Keras* and *Tensorflow* as shown in Figure 4.1. More than 81% of the API bugs are from *Keras* and *Tensorflow*. An example of such bug is shown in the code snippet below. The bug in the code below arises because the keyword names in the API signature of *Keras* has changed.

```
model.fit(tX, tY, epochs=100, batch_size=1, verbose=2)
```

The developer will get the error because `epochs` keyword does not exist in version 2+ of *Keras*.

```
model.fit(tX, tY, batch_size=1, verbose=2, epochs = 100) File
"keras/models.py", line 612, in fit str(kwargs))
Exception: Received unknown keyword arguments: {'epochs': 100}
```

To fix this error, the developer needs to change the keyword parameter from `epochs` to `nb_epoch`.

Table 4.2: Statistics of Bug Types in *Stack Overflow* and *Github*

	Caffe		Keras		TF		Theano		Torch		P value
	SO	GitHub	SO	GitHub	SO	GitHub	SO	GitHub	SO	GitHub	
API Bug	6%	0%	11%	57%	11%	72%	7%	3%	16%	2%	0.3207
Data Bug	9%	49%	24%	8%	30%	0%	35%	17%	36%	15%	0.3901
NMSB.Control and Sequence Bug	0%	8%	0%	0%	0%	0%	4%	0%	0%	7%	0.3056
NMSB.Initialization Bug	0%	0%	1%	0%	1%	0%	0%	3%	0%	0%	0.7655
NMSB.Logic Bugs	11%	0%	13%	2%	8%	0%	25%	6%	12%	7%	0.0109
NMSB.Processing Bug	0%	0%	0%	0%	1%	0%	0%	3%	0%	7%	0.2323
SB.Control and Sequence Bug	6%	12%	2%	0%	4%	0%	4%	3%	8 %	9%	1.0000
SB.Data flow Bug	3%	8%	13%	26%	15%	0%	0%	14%	4%	16%	0.2873
SB.Initialization Bug	0%	0%	1%	0%	8%	1%	0%	23%	20%	11%	0.8446
SB.Logic Bugs	42%	15%	27%	3%	18%	23%	18%	14%	0%	13%	0.3442
SB.Processing Bug	23%	8%	8%	4%	4%	4%	7%	14%	4%	13%	0.8535

```
model.fit(tX, tY, nb_epoch=100, batch_size=1, verbose=2)
```

4.3.4 Bugs in *Github* Projects

We have also analyzed the distributions of bugs in some *Github* bug fix commits. The distribution of bugs across different libraries in *Github* data is shown in Table 5.3. We computed the P value using t-test where one distribution is bug type in *Github* for all the libraries and the other distribution is bug type for all the libraries in *Stack Overflow*.

Finding 4 ⇒ All the bug types have a similar pattern in *Github* and *Stack Overflow* for all the libraries

We analyze the *Stack Overflow* and *Github* result using the t-test to find whether the distributions differ significantly. We use 95% significant level to find the difference beween *Stack Overflow* and *Github* results

for each of the bug type In our analysis the null hypothesis is: ***H_0: The distributions are same***. If we fail to reject this null hypothesis using the t-test then we can say the distributions follow the same pattern in both *Stack Overflow* and *Github* data.

We see that for all the bug types except Non Model Structural Logic Bug the P value is greater than 5% indicating they have a similar pattern as we fail to reject the null hypothesis.

4.4 Root Cause

In this section, we present the analyses and findings to answer **RQ2** identifying major root causes of bugs in deep learning software. The normalized distribution of root causes in *Stack Overflow* code snippets is shown in Figure 4.2. The data in Table 4.3 shows the presence of different categories of root causes in both *Stack Overflow* and *Github* for the deep learning libraries and presents P value showing the similarity of distributions using t-test. We discuss the significant root causes in the following subsections.

4.4.1 Incorrect Model Parameter (IPS)

Finding 5 ⇒ IPS is the most common root cause resulting in average 24% of the bugs across the libraries

IPS results in bugs that causes the program to crash at runtime and the execution does not succeed. In *Tensorflow* and *Theano* IPS leads other root causes in causing bugs having 26% and 26% of the total share of root causes, respectively.

4.4.2 Structural Inefficiency (SI)

Finding 6 ⇒ *Keras*, *Caffe* have 25% and 37% bugs that arise from SI

SI bugs do not cause the program to crash. These bugs often yield suboptimal performance of the deep learning model. These bugs have more relation to QoS or non-functional requirements. For example, a

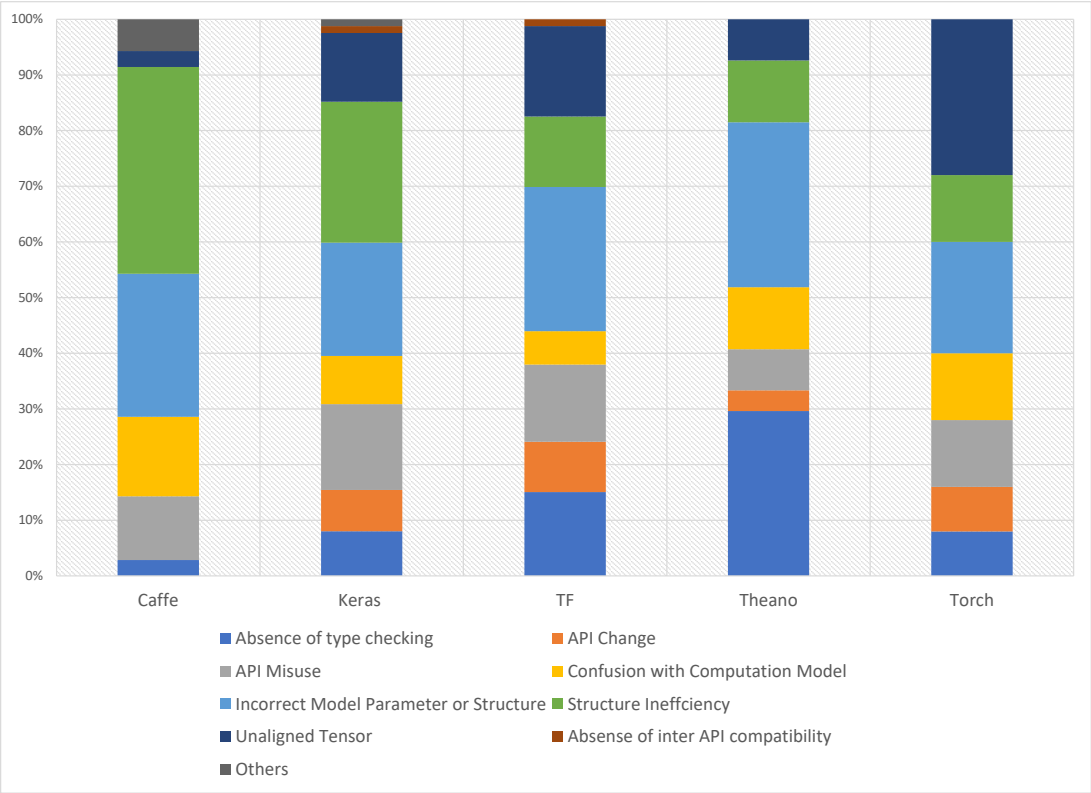


Figure 4.2: Stack Overflow Root Cause Classification

programmer is trying to train a model to recognize handwritten digits but the accuracy does not improve and stays constant from epochs 2 - 10.[2]

```
Epoch 1/10
2394/2394 [==============================] - 0s - loss: 0.6898 - acc: 0.5455 -
    val_loss: 0.6835 - val_acc: 0.5716
Epoch 2/10
2394/2394 [==============================] - 0s - loss: 0.6879 - acc: 0.5522 -
    val_loss: 0.6901 - val_acc: 0.5716
.........
```

[2] https://stackoverflow.com/questions/37213388/keras-accuracy-does-not-change

```
Epoch 10/10
2394/2394 [==============================] - 0s - loss: 0.6877 - acc: 0.5522 -
    val_loss: 0.6849 - val_acc: 0.5716
1027/1027 [==============================] - 0s
```

The problem that was pointed out by an expert, which solved the performance degradation bug is following:

```
#In summary, replace this line:
model.compile(loss = "categorical_crossentropy", optimizer = "adam")
#with this:
from keras.optimizers import SGD
opt = SGD(lr=0.01)
model.compile(loss = "categorical_crossentropy", optimizer = opt)
```

The answer suggested to change `optimizer` for enhancing the performance.

4.4.3 Unaligned Tensor (UT)

Finding 7 ⇒ *Torch* has 28% of the bugs due to UT

In deep learning, tensor dimensions are important for successful construction of the model. *Tensorflow*, *Keras*, *Torch*, *Theano*, *Caffe* have 16%, 12%, 28%, 7% and 3% of bugs due to UT respectively. In *Torch* UT is the leading root cause of bugs.

4.4.4 Absence of Type Checking

Finding 8 ⇒ *Theano* has 30% of the bugs due to the absence of type checking

Most of the deep learning libraries are written in Python. Due to the dynamic nature of Python, the problem of the absence of type checking is felt strongly in these libraries. The absence of type checking leads to 30% of the bugs in *Theano*, 8% of the bugs in *Keras* and 15% of the bugs in *Tensorflow*.

4.4.5 API Change

Finding 9 ⇒ *Tensorflow* and *Keras* have 9% and 7% bugs due to API change

In deep learning libraries, API change tends to have a drastic effect. These libraries are interdependent. So, API change in one library breaks other libraries.

4.4.6 Root Causes in *Github* Data

Finding 10 ⇒ Except API Misuse all other root causes have similar patterns in both *Github* and *Stack Overflow* root causes of bugs

We computed the P value at 95% significant level for both the *Stack Overflow* and *Github* data for all the root causes in the five libraries. We see that, P value for API Misuse root cause is much less than 5% indicating API Misuse in *Stack Overflow* and *Github* has different distribution compared to other root causes as we reject the null hypothesis. The other root causes are similar for both *Stack Overflow* and *Github* data as their P value is greater than 5%.

4.4.7 Relation of Root Cause with Bug Type

Finding 11 ⇒ SI contributes 3% - 53% and IPS contributes 24% - 62% of the bugs related to model

We have seen from Figure 4.3 that most of the non model related bugs are caused by API Misuse (6% - 100%). Non Model Structural Initialization Bugs and Non Model Structural Processing Bugs are caused by API Misuse in 100% of the time in our studied data. Interestingly in API Bug API Change plays the vital role (68%) compared to API Misuse (20%); however, the model related bugs are more vulnerable to IPS and SI root causes. We see from Figure 4.3 that Structural Control and Sequence Bug, Structaral Data Flow Bug, Structural Initialization Bug, Structural Logic Bug, Structural Processing Bug which are related to model are caused by SI 31%, 3%, 10%, 33% and 53% of the times respectively and caused by IPS 62%, 59%, 40%, 36%, 24% of the times respectively.

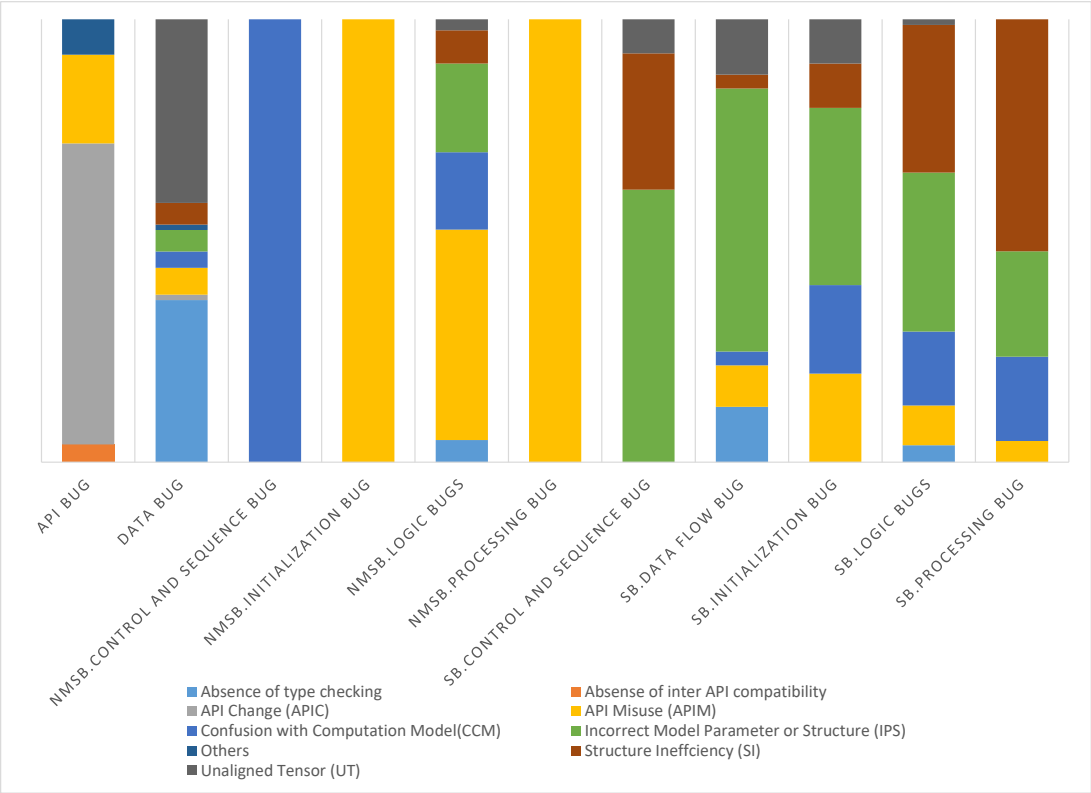


Figure 4.3: Relation between Root Causes and Types of Bugs

4.5 Impacts from Bugs

In this section, we explore the answer to **RQ3** to understand the major effects of bugs in deep learning software. The normalized distribution of effects of *Stack Overflow* is shown in Figure 4.4. The data in Table 4.4 shows the presence of different kinds of effects in both *Stack Overflow* and *Github* for the deep learning libraries. We discuss some of the major effects of bugs in deep learning software in the rest of this section.

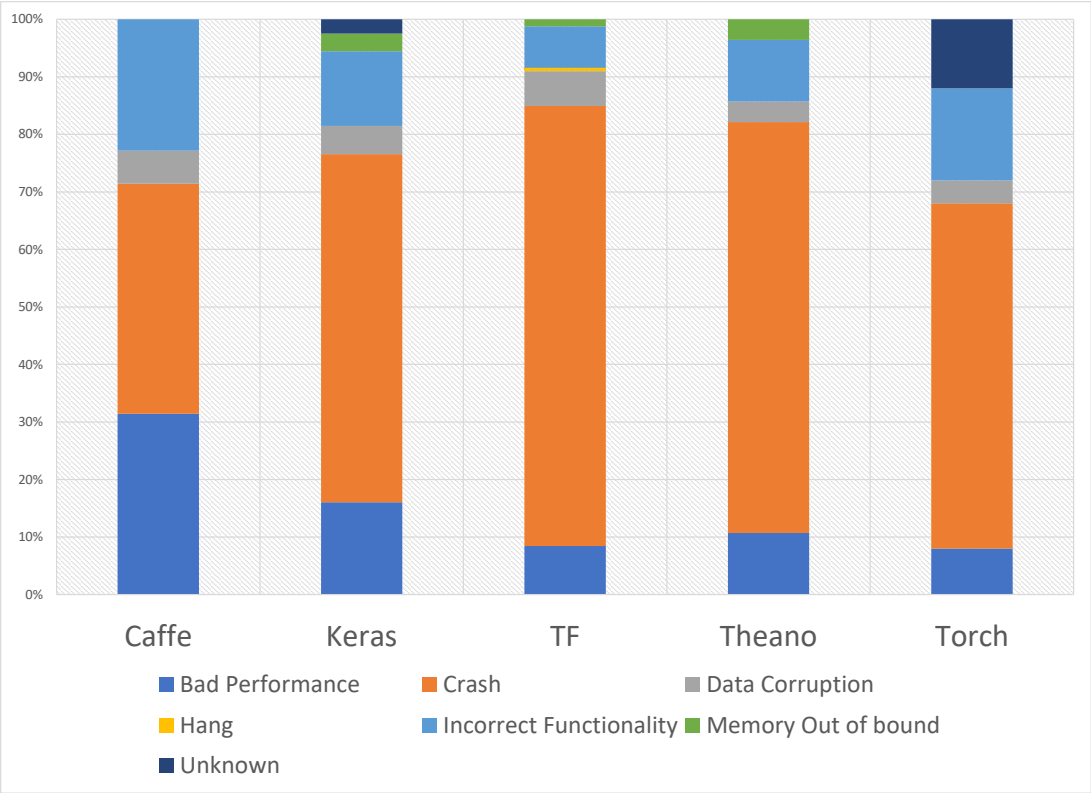


Figure 4.4: Distribution of Bug Effects in *Stack Overflow*

4.5.1 Crash

Finding 12 ⇒ More than 66% of the bugs cause crash.

Our analysis reveals that, the most severe effect of bugs is Crash. In deep learning, the bugs mostly cause total failure of the program. In all the libraries Crash is the top impact ranging from 40% - 77% as shown in Figure 4.4.

4.5.2 Bad Performance

Finding 13 ⇒ In *Caffe*, *Keras*, *Tensorflow*, *Theano*, *Torch* 31%, 16%, 8%, 11%, and 8% bugs lead to bad performance respectively

Bad performance is often a concern for deep learning software developers. Even though the model trains successfully, during the evaluation or prediction phase the model may give very poor accuracy in classifying the target classes.

For example, in the following code snippet the user had low accuracy after training because of the use of incorrect value of parameter `nb_words` that is the value of the maximum size of the vocabulary of the dataset. The developer should use `nb_words + 1` instead of `nb_words` as answered by an expert [3]. If the developer uses `nb_words` instead of `nb_words + 1`, the model will not train on the last word, which can lead to the bad performance effect.

```
embedded = Embedding(nb_words, output_dim=hidden, input_length=maxlen)(sequence)
```

4.5.3 Incorrect Functionality

Finding 14 ⇒ 12% of the bugs cause Incorrect Functionality

Incorrect functionality happens when the runtime behavior of the software leads to some unexplainable outcome that is not expected from the logical organization of the model or from previous experience of the developer.

For example, in the following code snippet the user wants to convert the image to a $28*28$ `Numpy` array; however, the output is a black image.[4]

```
with tf.Session() as sess:
    first_image = mnist.train.images[0]
    first_image = np.array(first_image, dtype='uint8')
    pixels = first_image.reshape((28, 28))
```

[3] https://stackoverflow.com/questions/37817588/masking-for-keras-blstm
[4] https://stackoverflow.com/questions/42353676/display-mnist-image-using-matplotlib

```
5     plt.imshow(pixels, cmap='gray')
```

The user got incorrect output because of casting a float array to uint8, which will convert all the pixels to 0 if they are less than 1. To fix the problem, the user can multiply the array with 255 as suggested by an answer. *Theano* has a higher percentage of posts about incorrect functionality problems compared to bad performance.

4.5.4 Effects of Bugs in *Github*

Finding 15 ⇒ For all the libraries the P value for *Stack Overflow* and *Github* bug effects reject the null hypothesis to confirm that the bugs have similar effects from *Stack Overflow* as well as *Github* bugs

The P value is shown in Table 4.4 shows that Bad Performance in *Stack Overflow* and *Github* have 79% of P value which indicates that they are very similar. Crash has P value of 50% in *Stack Overflow* and *Github* indicating they also can not reject the null hypothesis with strong confidence. None of the impacts reject the null hypothesis at 95% significance level.

4.6 Difficult Deep Learning stages

In this section, we answer **RQ4** by studying the bugs arising at the different stage of the deep learning pipeline. We use the categorization of the posts about deep learning stages to analyze **RQ4**.

4.6.1 Data Preparation

Finding 16 ⇒ 32% of the bugs are in the data preparation stage

From Figure 4.5 we see, most of the bugs in deep learning programming happen at the data preparation stage.

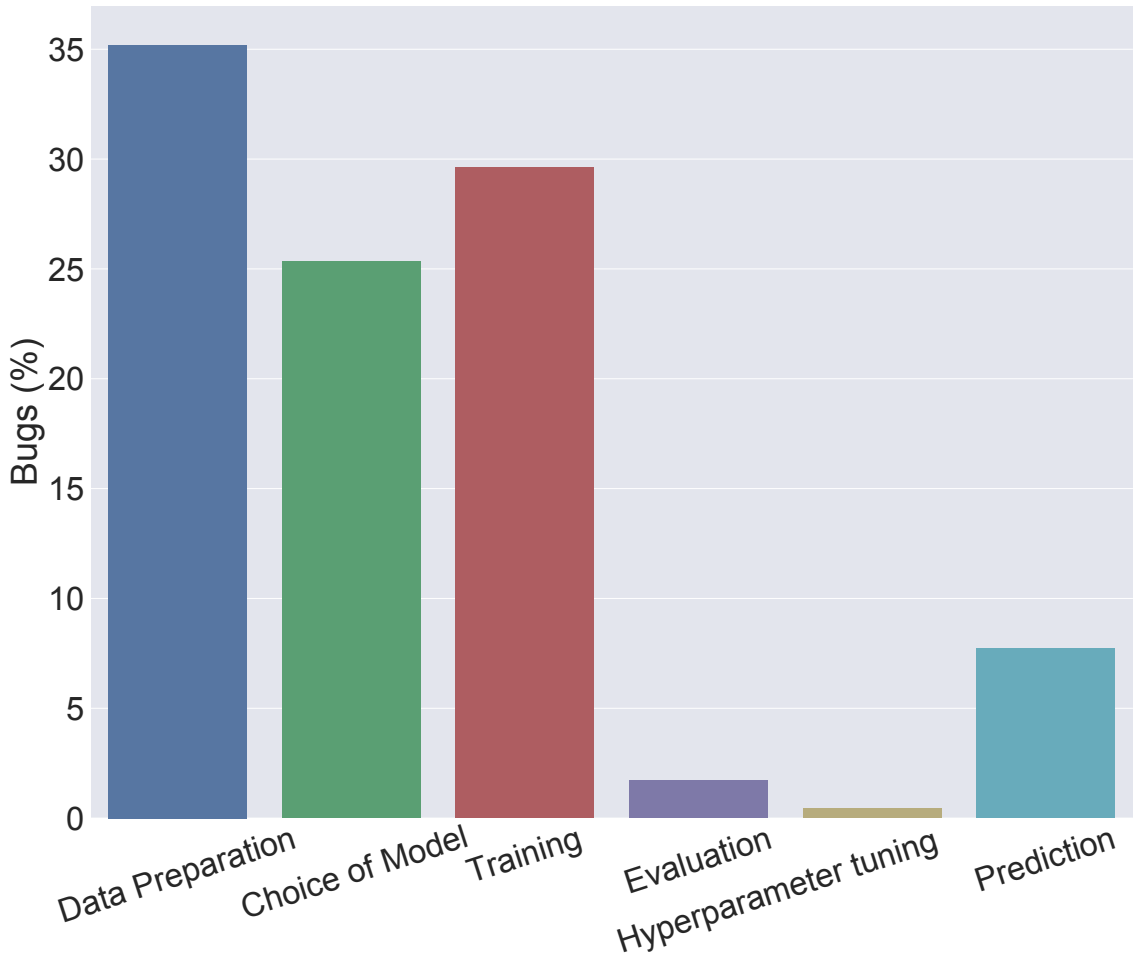


Figure 4.5: Bugs across stages of the Deep Learning pipeline

4.6.2 Training Stage

Finding 17 $\Rightarrow$ 27% of the bugs are seen during the training stage

The next bug prone stage is the Training stage which is as expected. Most bugs related to IPS and SI arise in the training stage.

4.6.3 Choice of Model

Finding 18 ⇒ Choice of model stage shows 23% of the bugs

Choice of model is the third bug prone stage. In choice of model stage, we construct the model and chose the right algorithm. Major root causes of bugs in this stage are IPS, SI, and UT.

4.7 Commonality of Bug

In this section, we explore the answer to **RQ5** to identify whether there is any relationship among the bugs in different deep learning libraries. Our primary hypothesis was that the libraries will be strongly correlated based on the distribution of bugs as they are performing similar tasks.

Our analysis confirms that hypothesis as shown in Figure 4.6. We see that the libraries have a strong correlation coefficient close to 1. Surprisingly *Caffe* has shown very weak correlation with other libraries in terms of bug type. We then randomly studied 30 *Stack Overflow* posts for each of the libraries to see whether we notice any common antipatterns that can lead to this strong correlation of bug type.

Finding 19 ⇒ *Tensorflow* and *Keras* have a similar distribution of antipatterns while *Torch* has different distributions of antipatterns

We have identified the antipatterns through deeper analysis of the *Stack Overflow* buggy codes for further investigating the strong correlation of *Tensorflow* and *Keras* as well as the weak correlation of *Torch* and *Caffe*. The antipatterns found are **Continuous Obsolescence, Cut-and-Paste Programming, Dead Code, Golden Hammer, Input Kludge, Mushroom Management, Spaghetti Code**. This classification is taken from [111]. The distribution of different antipatterns across the libraries is shown in Figure 4.7. We see that in *Tensorflow* and *Keras* 40% of the antipatterns are Input Kludge. On the other hand, in *Torch* 40% of the bugs arise due to the Cut-and-Paste Programming antipattern. *Tensorflow* and *Keras* have almost same distribution in Continuous Obsolescence and Dead Code as well. This shows that the strong correlation between the distribution of bugs in *Tensorflow* and *Keras* can be explained from the similarity of common antipatterns for these two libraries. The weak correlation between the distribution of *Torch* and *Caffe* bugs

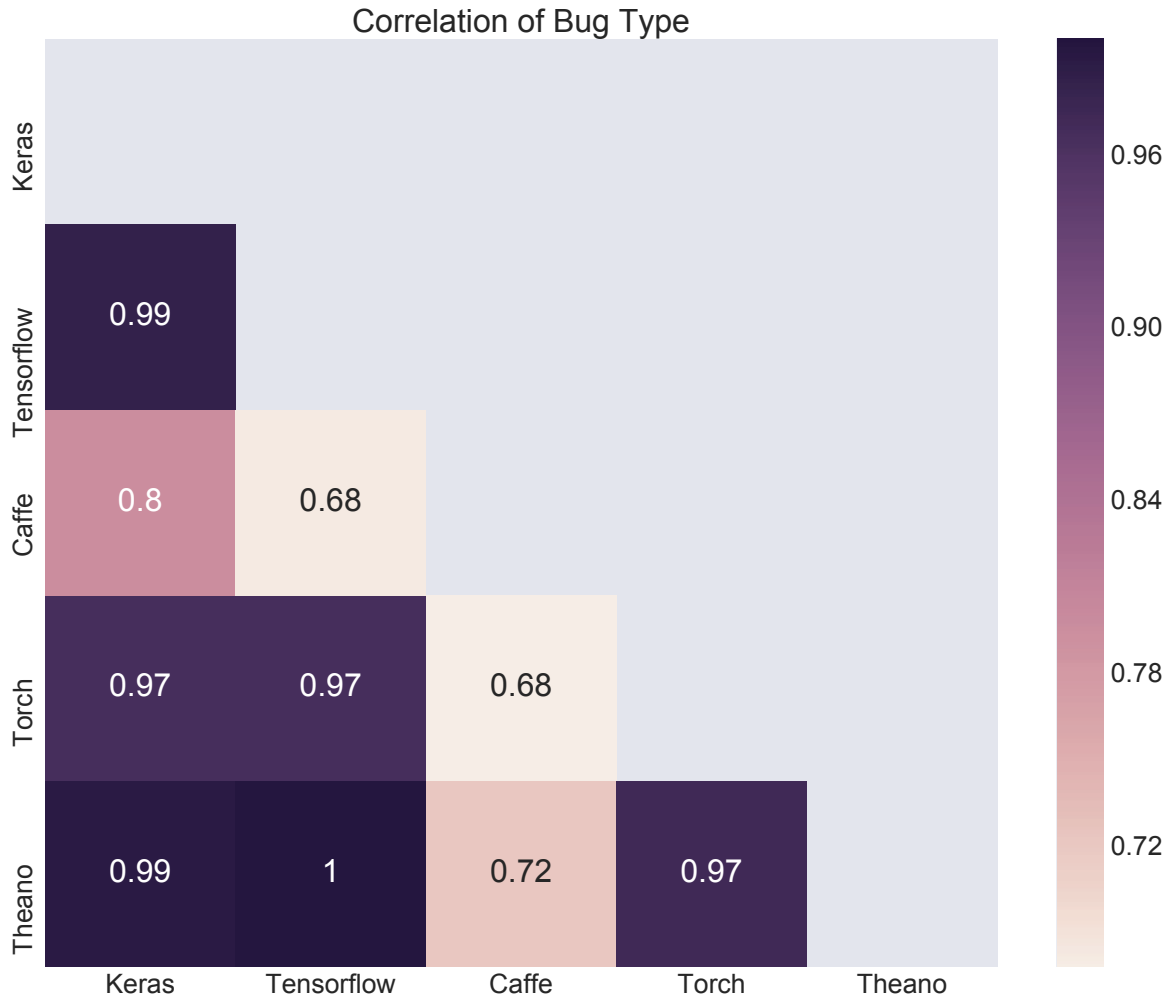


Figure 4.6: Correlation of Bug Types among the libraries

can be the result of a dissimilar distribution of antipatterns between these two libraries. For example, we see *Stack Overflow* code snippets of Input Kludge antipatterns from *Tensorflow* and *Keras* in the example shown in Figure 4.8. Both of these programs can be easily broken by user input and the program does not perform sanity check on the inputs.

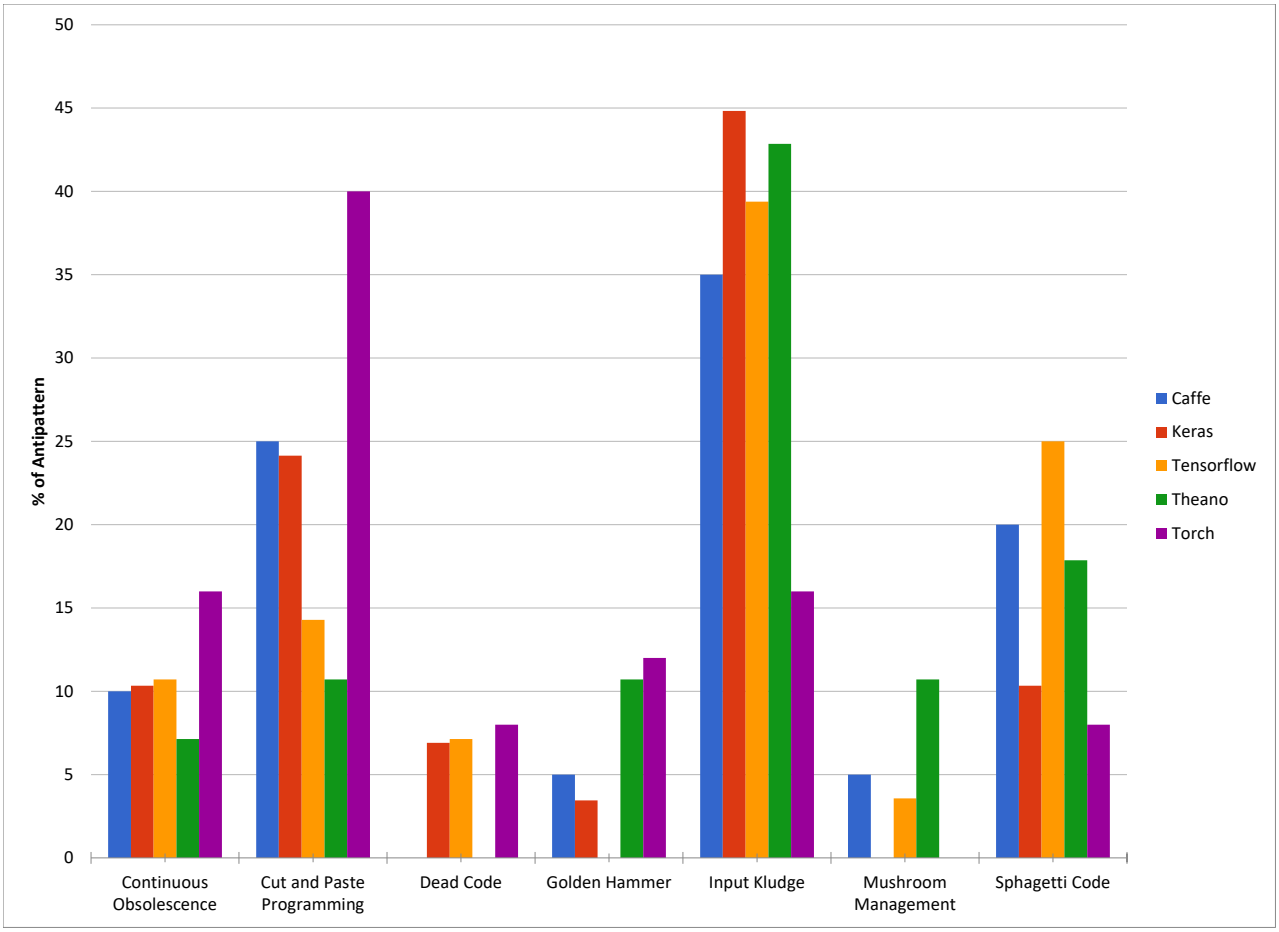
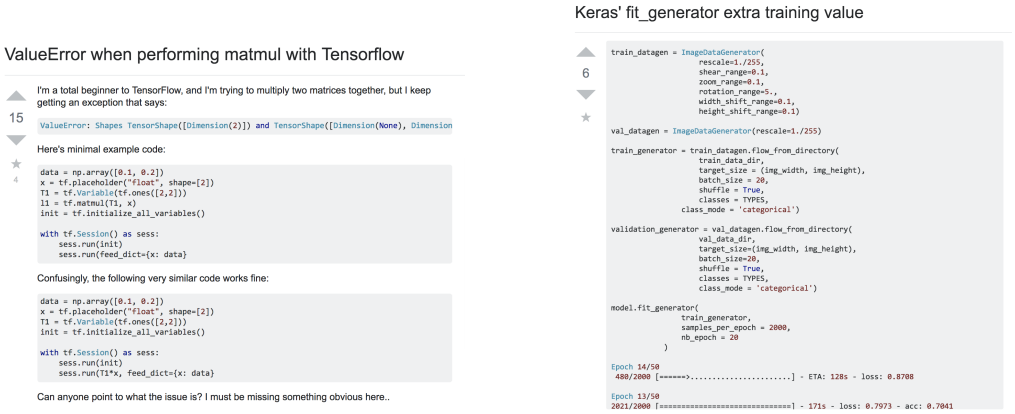


Figure 4.7: Distribution of different antipatterns

4.8 Evolution of Bugs

In this section, we explore the answer to **RQ6** to understand how the bug patterns have changed over time.



(a) Tensorflow Example of Input Kludge (b) Keras Example of Input Kludge

Figure 4.8: Example of similar antipattern in *Tensorflow* and *Keras*

4.8.1 Structural Logic Bugs Are Increasing

Finding 20 ⇒ In *Keras*, *Caffe*, *Tensorflow* Structural logic bugs are showing increasing trend

From 2015 - 2018 Structural logic bugs in *Caffe* are respectively 30%, 32%, 67%, 100% indicating structural logic bugs are being discussed more by the developers since 2015. It is expected as deep learning started gaining increasing attention since 2015 and more developers started to use deep learning libraries to write software.

4.8.2 Data Bugs Are Decreasing

Finding 21 ⇒ Data Bugs slowly decreased since 2015 except *Torch*

In *Torch* Data Bugs stayed almost consistent maintaining close to 50% of the bugs in discussed in 2016-2018. In *Keras* Data Bugs slowly decreased from 27% - 15% since 2015. In *Tensorflow* Data Bugs slowly decreased from 30% - 10% since 2015 - 2018. In the other two libraries also, the Data Bugs slowly decreased reaching close to 0. The possible reason for this trend is the development of popular specialized data libraries like `pandas` that enable exploratory data analysis to understand the properties of data better. Besides, the use of Tensor data type having type and shape information helps to get rid of some of the Data Bugs. Still more verification support in these libraries will help to get rid of these bugs.

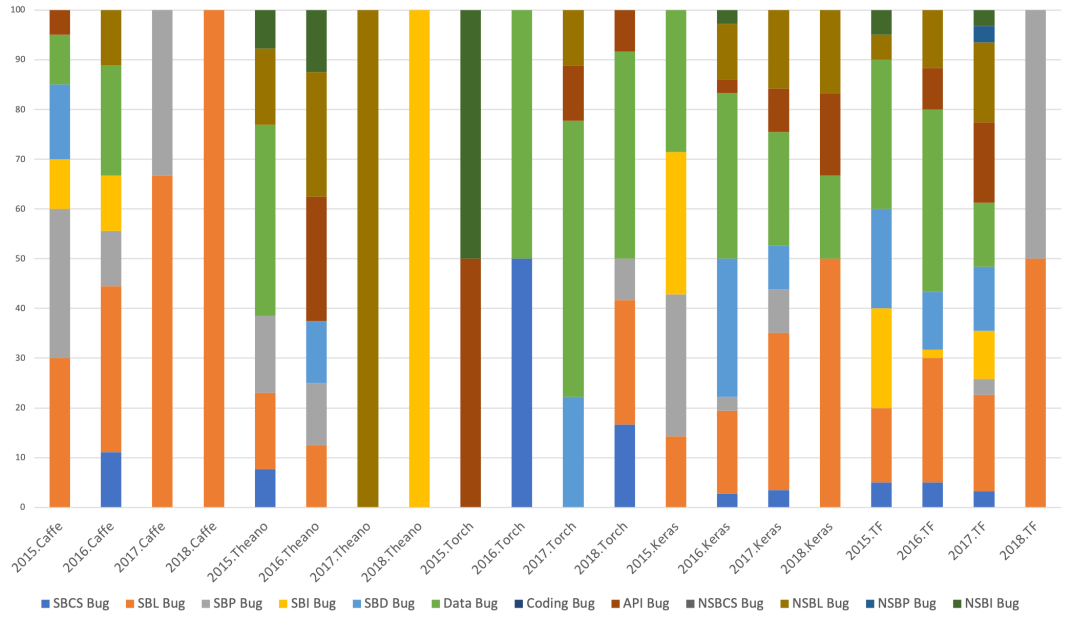


Figure 4.9: Timeline of Evolution of Bugs

4.9 Threats to Validity

Internal threat. One internal threat to the validity of our results could be our classification of the bugs. We used the classification scheme from a vetted taxonomy [51, 108] to classify the bugs. We also followed open coding scheme to add more types if needed. One PhD student was initially dedicated to go over all the posts to come up with additional classification scheme, if necessary. This whole process was monitored using pilot study. Another possible source of the threat is that the labeling of the data can be biased. To mitigate this threat two trained Ph.D. students independently studied the misuse posts to label them. The inter-rater agreements was measured using Cohen's Kappa coefficient and the disagreements were reconciled under the monitoring of an expert. We conducted pilot study to continuously monitor the labeling process and conducted further training at 5% and 10% of the labeling where the Kappa coefficient was close to 0% and 80%.

External threat. An external threat can be the trustworthiness of the dataset we collected. To avoid low-quality posts we only collected the posts that have score of at least 5. A score of 5 can be a good metric to trust the post as a good discussion topic among the programmer community that cannot merely be solved

using some Google search. The reputation of the users asking question about deep learning can be another reason to question the quality of the posts. To alleviate this threat we have only studied top scored posts which are from users with different range of reputations (1 - 150K+). This indicates that the posts are from users ranging from newbie to experts. The dataset is unbalanced in terms of frequency of bugs studied for each library; however, to confirm the distribution of the bugs, we have performed ANOVA test on the bug types, root causes, and impacts for each library. We have found that $F(0.99) < F\text{-}critical(2.55)$. This implies that the means of the five libraries population are not significantly different. This suggests that even though the dataset seems unbalanced in term of frequency, the bug distribution is not.

4.10 Discussion

We have seen in the analysis of RQ1 that most of the bugs in deep learning programming are Data Bugs. These type of Bugs can have drastic effect causing the program to crash as well as leading to bad performance. In general, we see the programmers have very limited or no access to data verification tools. It is often confusing whether the data is in right format needed by the model, whether the variables are properly encoded or not, whether there are missing data that can cause the model to fail, whether the train test split is good enough, whether the data is shuffled properly to avoid training bias etc. This finding suggests that development of **data verification tools** can help programmers solve a large number of data bugs. As deep learning models are strongly coupled with data, **model analysis** tool to explore whether a particular model is the right fit for the data in hand can help to resolve these strong coupling of data and model related problems.

We have also seen while exploring RQ1 that structural logic bugs are the second major type of bugs. This happens due to wrong logical organization of the model, hidden layers, using wrong codes, etc. These kind of problems can be solved by some **automated model and parameter recommendation** tools. How to develop these kind of tools need further research. A methodology could be to mine large scale open source code repositories [112–114] using Python dataset [7] and identify the common code patterns and suggest examples from common code patterns.

4.11 Conclusion

Although deep learning has gained much popularity and strong developer community in recent years, developing software using existing deep learning libraries can be error-prone. In this chapter, we have presented an empirical study to explore the bugs in software using deep learning libraries. In our study we have studied 2716 qualified *Stack Overflow* posts and 500 *Github* bug fix commits to identify the bug types, root causes of bugs, effects of bugs in usage of deep learning. We have also performed an inter-stage analysis to identify the stages of deep learning pipeline that are more vulnerable to bugs. We have also studied the buggy codes in *Stack Overflow* to find antipatterns leading to bugs to understand the strong correlation of the bug types in deep learning libraries. Our study found that data bug and logic bug are the most severe bug types in deep learning software appearing more than 50% of the times. Major root causes of these bugs are Incorrect Model Parameter (IPS) and Structural Inefficiency (SI). Last but not least, bugs in the usage of deep learning libraries are strongly correlated. This work opens multiple avenues for exploration. For instance, while we have studied bugs, we haven't yet examined the fix strategies that programmers use. This study is also on a relatively modest dataset and could be repeated on a much larger dataset. Finally, repair strategies could be developed for deep learning programs.

Table 4.3: Statistics of the Root Causes of Bugs

	Caffe		Keras		TF		Theano		Torch		P value
	SO	GitHub	SO	GitHub	SO	GitHub	SO	GitHub	SO	GitHub	
Absense of inter API compatibility	0%	0%	1%	0%	1%	0%	0%	0%	0%	0%	0.1411
Absence of type checking	3%	12%	8%	3%	15%	15%	30%	20%	8%	13%	0.9717
API Change	0%	0%	7%	51%	9%	58%	4%	0%	8%	2%	0.2485
API Misuse	11%	0%	15%	4%	14%	0%	7%	3%	12%	2%	0.0003
Confusion with Computation Model	14%	28%	9%	1%	6%	10%	11%	3%	12%	4%	0.7839
Incorrect Model Parameter or Structure	26%	31%	21%	30%	26%	16%	30%	14%	20%	19%	0.5040
Others	0%	0%	0%	0%	0%	0%	0%	0%	0%	2%	0.3466
Structure Inefficiency	37%	12%	26%	5%	13%	1%	11%	26%	12%	38%	0.7170
Unaligned Tensor	3%	19%	12%	5%	16%	0%	7%	34%	28%	20%	0.7541
Wrong Documentation	6%	0%	1%	1%	0%	0%	0%	0%	0%	0%	0.3402

Table 4.4: Effects of Bugs in *Stack Overflow* and *Github*

	Caffe		Keras		TF		Theano		Torch		P value
	SO	GitHub	SO	GitHub	SO	GitHub	SO	GitHub	SO	GitHub	
Bad Performance	31%	19%	16%	14%	8%	8%	11%	6%	8%	24%	0.9152
Crash	40%	69%	61%	86%	77%	92%	70%	20%	60%	16%	0.7812
Data Corruption	6%	4%	5%	0%	6%	0%	4%	6%	4%	16%	0.948
Hang	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%	0.3466
Incorrect Functionality	23%	8%	13%	0%	7%	0%	11%	59%	16%	42%	0.5418
Memory Out of bound	0%	0%	3%	0%	1%	0%	4%	0%	0%	0%	0.0844
Unknown	0%	0%	2%	0%	0%	0%	0%	9%	12%	2%	0.8419

CHAPTER 5. REPAIRING DEEP NEURAL NETWORKS: FIX PATTERNS AND CHALLENGES

5.1 Introduction

The availability of big data has fueled the emergence of deep neural networks (DNN). A DNN consists of a set of layers. Each layer contains a set of nodes collecting inputs from the previous layer and feeding the output to nodes in the next layer via a set of weighted edges. These weights are adjusted using examples, called training data, and set to values that minimize the difference between actual outputs of the DNN and expected outputs measured using an objective function called loss function. The availability of big data has made it possible to accurately adjust weights for DNNs containing many layers. Thus, many software systems are routinely utilizing DNNs. SE for DNNs has thus become important.

A significant SE problem in the software that uses DNNs is the presence of bugs. What are the common bugs in such software? How do they differ? Answering these questions has the potential to fuel SE research on bug detection and repair for DNNs. Fortunately, recent work has shed some light on this issue. Zhang *et al.* [51] have identified bug types, root causes, and their effects in *Tensorflow* library for DNN. Islam *et al.* [3] have studied an even larger set of libraries including *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch* to identify bug characteristics. While prior work presents an initial study on repair patterns for *Tensorflow*, these works have not focused on the characteristics of repairs. Since repairing software that uses DNNs is an unmistakable SE need where automated tools could be very helpful, fully understanding the challenges to repairing and patterns that are utilized when manually repairing bugs in DNNs is critical. What challenges should automated repair tools address? What are the repair patterns whose automation could help developers? Which repair patterns should be prioritized?

Motivated by these questions, we conduct a comprehensive study of bug repair patterns for five DNN libraries *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch*. We leverage the dataset of DNN bugs published by Islam *et al.* [3] that consists of 415 bugs from *Stack Overflow* and 555 bugs from *Github*. We then collect

the code snippets used to fix these bugs from both *Stack Overflow* and *Github*. We then manually study these repairs and label them according to a classification scheme developed using the open coding approach. To study the fix in *Stack Overflow* we study the accepted answers and answers with score >= 5 from *Stack Overflow* post that fixes the bug in the original post. To study the bug fix patterns in *Github*, we take the bug-fix commits in the dataset and study the code that is changed to fix the bug. If we do not find any fixes that match our selection criteria in *Stack Overflow* and relevant fix in *Github* we discard those bugs. In total, we have studied *320* bug fix codes in *Stack Overflow* and *347* bug fix codes in *Github*. We have also analyzed these bug fixes to answer the following research questions:

RQ1 **(Common bug fix patterns)** What are the most common bug fix patterns?

RQ2 **(Fix pattern across bug types)** Are the bug fix patterns different for different bug types?

RQ3 **(Fix pattern across libraries)** Are the bug fix pattern different for different libraries?

RQ4 **(Risk in fix)** Does fixing a DNN bug introduces a new bug?

RQ5 **(Challenges)** What are the challenges in fixing DNN bugs?

Our key findings are as follows: DNN bug fix patterns are distinctive compared to traditional bug fix patterns; the most common bug fix patterns are fixing data dimension and network connectivity; DNN bug fixes have the potential to introduce adversarial vulnerabilities [115]; DNN bug fixes frequently introduce new bugs; and DNN bug localization, reuse of trained model, and coping with frequent releases are major challenges faced by developers when fixing bugs. We also contribute a benchmark of 667 DNN (bug, repair) instances. This benchmark is also publicly accessible [116].

5.2 Methodology

5.2.1 Dataset

In our study, we build on the bug dataset prepared by Islam *et al.* [3] to collect and to prepare the dataset of bug fixes. The bug dataset contains 415 bugs from *Stack Overflow* and 555 bugs from *Github* for 5 different deep learning libraries as shown in Table 5.1.

Table 5.1: Summary of the bug repair dataset.

Library	*Stack Overflow*		*Github*	
	Bugs [3]	Fixes (current)	Bugs [3]	Fixes (current)
Caffe	35	**27**	26	**17**
Keras	162	**143**	348	**167**
Tensorflow	166	**118**	100	**90**
Theano	27	**15**	35	**32**
Torch	25	**17**	46	**41**
Total	415	**320**	555	**347**

Collecting *Stack Overflow* bug fixes: To collect the bug fixes in *Stack Overflow* bug dataset, we study all the answers corresponding to the post ids in *Stack Overflow* bug dataset. If a post has accepted an answer with code, then we consider that code snippet as a fix. If the accepted answer doesn't have code but describes what needs to be fixed in the original bug we consider those as fix as well. If a bug post does not have an accepted answer but has an answer with >= 5 scores we consider them as fixes also as score 5 is considered as an acceptable quality metric in prior works [3]. Following this methodology, we were able to find 320 fixes for 320 bug related posts in the *Stack Overflow* dataset.

Collecting *Github* bug fixes: To collect *Github* bug fixes, we went to the link of the buggy code snippets in the dataset. If the code snippet was fixed in a later revision, then we take those fixes. A single line may contain multiple bugs [3]. A single bug fix commit might fix multiple bugs. We consider them different fixes. For example, in the same fix API name is updated from deprecated to a new version and the dimension is also fixed. We consider them as two different fixes. Some of the bugs are not yet fixed in the repositories and some repositories have been made private or deleted since the previous study. We omitted those bugs. Following this methodology, we collected 347 bug fixes from *Github*.

5.2.2 Bug Fix Pattern Classification

Next, we created a classification scheme to manually label the bug fix dataset. We started with the classification scheme used by Pan, Kim, and Whitehead [41] and found that their classification scheme has 28 non-ML bug fix categories and among them only 4 fix categories are applicable for the DNN-related fixes. Then, we used the open coding approach to refine it to come with a pattern of 15 different kinds of

Table 5.2: Summary of the bug fix patterns.

Bug Fix Pattern	Definition
Loss function	add, remove or replace the loss function.
Network connection	change node connectivity in the DNN, e.g. change weights, remove edges, add backward propagation.
Add layer	add another layer to the DNN model
Layer dimension	change a layer's input and output size, e.g. to make it compatible with adjacent layers' dimension
Data dimension	align the input data's dimension with the layer dimension
Accuracy metric	replace the accuracy metric being used to measure the correctness of a model, often to match better
Data type	change the type of data given as input to the DNN
Activation	change the activation function used in the DNN
Iterations	change the number of times the training would be done, e.g. modify batch size, epoch or add a loop
Versioning	adapt the code to the new version of the library
API contract	fix API compositions so that the output of an API meets the preconditions of another API
Data wrangling	fix the form of the data for downstream operations without modifying its intent
Monitor	add diagnostics code to monitor training
Optimizer	change the optimization function used by the DNN
Change neural architecture	overhaul the design of the DNN's architecture including a new set of layers and hyperparameters, generally because changes above can't fix the bug

DNN-specific bug fix patterns. We conducted a pilot study where two Ph.D. students individually studied the fixes to come up with a possible classification. Each student proposed a set of classes that were then reconciled during an in-person meeting where all the authors were present. In the in-person meeting, the authors validated the classification schemes from two individual raters and updated the classification scheme based on the outcome of the reconciliation effort under the supervision of the moderator. Our pilot study revealed that there are a number of unique bug fix patterns in our DNN setting. Therefore, the classification from prior work had to be significantly modified. The final classification is shown in Table 5.2 and discussed below.

Finding 1 ⇒ We found that DNN bug fix patterns are very different from traditional bug fix patterns such as [41].

5.2.2.1 Loss Function

This group of fixes is based on the addition, removal, or update of the loss function during training. The loss function is a key parameter that helps the training process to identify the deviation from the learned and actual examples. Different kind of problems demand a different loss function, e.g., cross-entropy loss is widely used in the classification problems whereas mean square error loss (MSE) is mostly used for regression-based problems. Some problems ask for a custom loss function for better training result and we group this kind of fixes into this class.

5.2.2.2 Network Connection

This group of fixes changes the connection between nodes in the DNN. A DNN is a graph, where edges are the weights and bias and nodes are the elements of each layer. For example, in a dense layer, the weight edges are fully connected with the next layer and the dimension of the layer determines the number of nodes to be available in that layer. Those bug fixes that reconfigure these connections for better results are classified in this category. The changes include change of weight, removing edges by pruning the network, adding backward propagation, etc.

5.2.2.3 Add Layer

In any classification based problem, there will be at least two layers in the model, the input layer, and the output layer. To learn the features of the input, a DNN frequently needs more intermediate layers (called hidden). This group of fixes adds more layers to the DNN to improve performance. Added layers can be dense, where two consecutive layers are fully connected, convolution layer, where convolution function has been applied to the input, dropout layer for reducing the overfitting, etc.

5.2.2.4 Layer Dimension

These fixes change the dimensions of the layers to make them compatible with adjacent layers and input.

5.2.2.5 Data Dimension

Data dimension related fix is similar to layer dimension, but it is related to the input data rather than to the DNN layers. The dimension of the data needs to be aligned with the DNN. This type of fix is mostly needed when the input dimensions of the DNN and the data dimension do not match.

5.2.2.6 Accuracy Metric

To measure the correctness of a DNN, the accuracy metric is one of the key parameters to be configured. The problem type has a huge influence on the type of accuracy metric to be used, e.g., classification problems are judged using classification accuracy, F1 score or confusion matrix, but these metrics are unsuitable for assessing a regression-based model where logarithmic loss is more suitable.

5.2.2.7 Data Type

This group of fixes changes the data type of inputs to match the DNN's expectation.

5.2.2.8 Activation

The activation function for a node in a layer of DNN maps inputs to the output. This group of fixes changes the activation function used in a layer to better match the problem.

5.2.2.9 Iterations

This group of fixes adjusts the number of times the training process will be run.

This is generally done to improve accuracy or to reduce overfitting. These fixes include changing batch size or epoch. In some cases, developers add a loop around the entire training process.

5.2.2.10 Versioning

DNN libraries are being rapidly developed, and a number of releases are not backward compatible that breaks code. This group of fixes adapts a code to work with the new version of the DNN library.

5.2.2.11 API Contract

When the output of a DNN API is fed to the input of another DNN API operation, these two operations have to be compatible. This group of fixes adds adapters to fix incompatibilities between composed operations.

5.2.2.12 Data Wrangling

Data wrangling refers to changing the form of data without changing its intent. It is generally done to fix the data for the downstream operations. This group of fixes adds data wrangling to fix a DNN, e.g. by data shifting, shuffle, etc.

5.2.2.13 Monitor

The fixes in this category add code for diagnostics during the training process, typically to print training statistics. This group of fixes do not repair the flaw in the code, but they help to localize the bug.

5.2.2.14 Optimizer

This group of fixes modifies the optimization algorithms used by the DNN model. The optimization algorithm, which is dependent on the problem, determines the iterative process followed to improve the accuracy of the DNN model.

5.2.2.15 Change Neural Architecture

This group of fixes essentially re-do the DNN model because the initial model was unsuitable.

5.2.3 Labeling

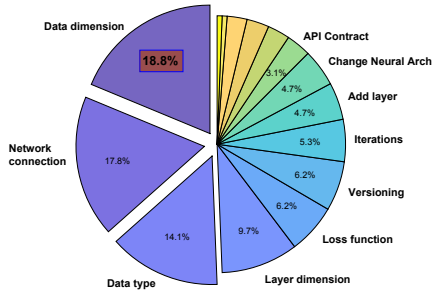
For labeling, we used the classification scheme shown in Table 5.2. Two Ph.D. students with expertise in these DNN libraries were requested to label the fixes according to the classification scheme. We held multiple training sessions to train the raters with the classification scheme. We used the Kappa coefficient [109] to measure the agreement between the raters after the labeling of every 100 bug fix patterns. We found that the Kappa coefficient was 82% for the first 100 labelings, 85% for the second 100 labeling. This high value of the Cohen's Kappa coefficient indicates perfect agreement between the raters. In the presence of a moderator, the repair patterns for which there was a label conflict between the raters were reconciled. We adapted this methodology from [3]. Following this strategy, we labeled all the fixes and reconciled the labeling conflicts through moderated discussions. The Kappa score throughout the process was >85% indicating a clear understanding and perfect agreement.

5.3 Bug Fix Patterns

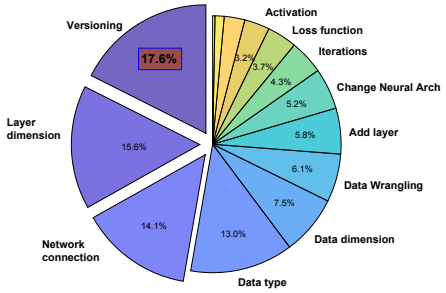
In this section, we explore the answer to RQ1 to understand what are the most common bug fix patterns in DNN. To answer RQ1, we take the labeled dataset and statistical distribution of the bug fix patterns across different categories. We also analyze the source code and diffs for those fixes to understand the challenges underlying those patterns. Figure 5.1 shows the distribution of different bug fix patterns in *Stack Overflow* and *Github*.

5.3.1 Data Dimension

Finding 22 ⇒ Fixing data dimension is the most common bug fix pattern (18.8%) in *Stack Overflow* that can affect the robustness of DNN model.



(a) Distribution of Bug Fix Patterns in *Stack Overflow* (Labels less than 3.1% are hidden)



(b) Distribution of Bug Fix Patterns in *Github* (Labels less than 3.2% are hidden)

Figure 5.1: Bug fix pattern distribution

Table 5.3: Bug Fixes in *Stack Overflow* (SO) and *Github* (GH)

	Caffe		*Keras*		*Tensorflow*		*Theano*		*Torch*	
	SO	GH	SO	GH	SO	GH	SO	GH	SO	GH
Loss function	11.1%	0.0%	6.3%	1.2%	4.2%	7.8%	13.3%	6.25%	5.9%	4.9%
Network connection	14.8%	11.8%	18.9%	10.2%	22%	13.3%	0.0%	21.9%	0.0%	26.8%
Add layer	11.1%	11.8%	5.6%	9.6%	2.5%	1.1%	0.0%	0.0%	5.9%	2.44%
Layer dimension	3.7%	0.0%	7.0%	26.3%	13.6%	3.3%	13.3%	9.4%	11.8%	9.8%
Data dimension	22.2%	0.0%	22.4%	9.6%	11.9%	2.2%	26.7%	15.6%	23.5%	7.3%
Accuracy metric	0.0%	0.0%	1.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Data type	3.7%	29.4%	7.7%	13.8%	19.5%	10.0%	26.7%	6.2%	29.4%	14.6%
Activation	7.4%	0.0%	3.5%	3.6%	0.8%	0.0%	6.7%	12.5%	0.0%	2.4%
Iterations	7.4%	5.9%	4.95%	3.6%	5.9%	4.4%	0.0%	9.4%	5.9%	2.4%
Versioning	0.0%	0.0%	6.3%	9.0%	8.5%	51.1%	6.7%	0.0%	5.9%	0.0%
API contract	3.7%	0.0%	2.1%	1.2%	5.1%	1.1%	0.0%	3.1%	0.0%	0.0%
Data wrangling	0.0%	35.3%	4.2%	2.4%	1.7%	1.1%	0.0%	6.2%	0.0%	19.5%
Monitor	11.1%	5.9%	2.8%	1.2%	1.7%	4.4%	0.0%	0.0%	0.0%	4.9%
Optimizer	0.0%	0.0%	1.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.4%
Change neural arch.	3.7%	0.0%	5.6%	8.4%	2.5%	0.0%	6.7%	9.4%	11.8%	2.4%

A large number of bugs (59 out of 415) in *Stack Overflow* are fixed by changing the data dimension. This suggests that most DNN models can easily be broken if the data processing pipeline changes or a different

format of data is fed to the DNN. For example, in the following code snippet, we see how the bug discussed in a *Stack Overflow* post[1] is fixed by adding a dimension to the input images.

```
model = Sequential()
...
model.compile()
model.load_weights('./ddx_weights.h5')
img = cv2.imread('car.jpeg', -1) # this is is a 32x32 RGB image
img = np.array(img)
+ img = img.reshape((1, 3, 32, 32))
y_pred = model.predict_classes(img, 1)
print(y_pred)
```

In the listing, the developer wants to read a CIFAR-10 image whose dimension is (32,32,3) but the expected image size was (1,3,32,32). Data dimension change can be categorized into the following kinds.

Resize Resizing the input data is common, e.g. resizing an input image of shape `(190,150)` to `(150, 150)`. A risk in this kind of fix is the loss of information from the input due to resizing. Surprisingly, this risk is never stated in the fixes presented on the bug fixes we have studied. 11 out of the 59 data dimension fixes involve resizing the data. Resizing can be done in two ways: downscale or upscale. The downscale is the method where the risk due to data loss is critical from our observation. Upsampling does not have this risk of data loss, and recent results suggest that adding noise to the data can potentially increase the robustness of a model [117].

Finding 23 ⇒ 63% of the resize related posts in *Stack Overflow* utilize the downscaling that can decrease the robustness of a DNN.

7 out of the 11 data resizing post in *Stack Overflow* involves downscaling. Downscaling decreases the robustness and [118] has shown that a simple resize downscaling operation can have a negative impact on the robustness. During downscaling, significant information loss occurs, and that eventually decreases the features learned by the DNN. A DNN trained with downscaled images will be easier to attack compared to

[1] https://stackoverflow.com/questions/37666887

the one trained with original images. Our findings suggest that it would be useful to verify the effect of the resizing fix on the vulnerability of the DNN.

Reshape Reshaping the input occurs when the input vector shape is changed. For example, a vector of size `(32, 32)` is changed to `(1,32,32)`. In this case, no data loss happens and the tensor order is changed from 2D to 3D. An example of this fix is presented in the *Stack Overflow* post #41563720[2]. The reshaping does not lead to data loss. 38 out of 59 data dimension fixes involve reshaping the dimension of the input. Reshape may also involve changing the dimension through one hot encoding like the following code snippet to fix *Stack Overflow* post #49392972[3]:

```
train_labels = to_categorical(train_labels)
```

Reorder To make this kind of dimension change, the input data is ordered mostly to change the channel position. In image classification problems, channel refers to the color channels of three primary colors. (height, width, channel) represents the typical structure of a 3D image. For example, the input of shape `(32,32,3)` is changed to `(3,32,32)` to fix some bugs. Here the channel number is moved to the first argument from the third argument. It can also involve changing the image dimension order format like from `RGB` to `BGR` as in the following snippet for fixing *Stack Overflow* post # 33828582[4]:

```
img = caffe.io.load_image( "ak.png" )
+ img = img[:,:,::-1]*255.0 # convert RGB->BGR
```

Finding 24 ⇒ Reorder and reshaping (79.7% of the data dimension fixes in *Stack Overflow*) need an understanding of the specifications of the DNN layers as well as the libraries.

9 out of 59 data dimension fixes involve reordering the dimension of inputs. This is done because some of the libraries require dimension in a specific order. These fixes are seen in the bugs where the developer works with multiple libraries having different channel position requirements in the image data, such as *Stack*

[2]https://stackoverflow.com/questions/41563720
[3]https://stackoverflow.com/questions/49392972
[4]https://stackoverflow.com/questions/33828582

Overflow post #45645276[5]. DNN training can be assumed as a gradient descent based optimization problem, which can be computed when all the functions utilized in the model creation are differentiable. Data should be changed in such a fashion that does not affect the gradient descent computation to avoid side effects. In *reshape* and *reorder*, the only changes occur is the addition of dimension and reordering of the values that do not impact the gradient descent computation. So these changes theoretically have no side effects in the DNN models' behavior.

5.3.2 Layer Dimension

Finding 25 $\Rightarrow$ In *Github* layer dimensions fixes are used more frequently (15.6%) to fix the crash related bugs (75.9%).

In *Github*, data dimension related fixes involve 7.5% of all the fixes. On the other hand, fixing the layer dimensions to make the DNN compatible with input data is a more common practice in *Github*. Dimension related fixes can be done by analyzing the input and output of the layers by converting a neural network into a data flow graph. This kind of fixes includes dimension reduction or addition based on the adjacent layers' structure. However, these fixes can be either done by changing the data dimension to match the data with the layer dimension or vice-versa. The choice of the fix has an impact on the performance of the model. This phenomenon is known as the *curse of dimensionality [119]*. The curse of dimensionality states that increasing or decreasing the dimension can lead to overfitting/underfitting problems. PCA [120], T-SNE [121] are some examples of the dimension reduction techniques that reduce the dimension of the features but these techniques suffer from the curse of dimensionality. To build an automated approach to avoid this side effect, a tool needs to optimize the performance of the model by either changing the data dimension or the layer dimension. AutoML [95] has done some preliminary work in this field that restructures the model by changing the layer dimension and adding layers to increase the performance. To the best of our knowledge, no tool currently exists that analyzes both data dimension and layer dimension changes to pick the optimum operations for a DNN model.

[5]https://stackoverflow.com/questions/45645276/

5.3.3 Version-related Fixes

Finding 26 ⇒ Versioning-related bug fixes are the highest (17.6%) in *Github* indicating the high maintenance cost in DNN software due to library versioning.

We have found that in *Github*, long-running projects have to fix a lot of bugs due to frequently changing versions of the DNN libraries. A number of these fixes require changing the API signatures to match with changes in the libraries. We have also observed a more complicated fix pattern for projects that use *Tensorflow* library as discussed in §5.7.3. *Tensorflow* often makes invasive, backward-incompatible changes adding difficulties to fix the introduced bugs. This indicates that the maintenance cost in DNN software is high.

5.3.4 Network Connection

Finding 27 ⇒ Network Connection is a prevalent fix in both *Stack Overflow* (17.8%) and *Github* (14.1%) to fix crash (57.14%), incorrect functionality (16.19%), and bad performance (12.38%) effects.

The tensor and data flow through the network in a DNN during forward and backward propagation or prediction. For a smooth flow of data, the end-to-end connectivity in the network is essential. 57 out of 415 fixes require fixing or adjusting the connectivity in the network. We have found three kinds of network connection repairs.

Merge layers A number of repair cases fixed bugs by merging two parallel layers into a single layer. For example, the following code snippet shows a fix,

```
+ main_branch.add(Merge([branch_1, branch_2], mode = 'dot'))
```

where two different branches are connected through dot product. The network was disconnected in the bug leading to a crash.

Add feedback loops and input layers In some bug fixes, a feedback loop is added in the DNN model. In some of the fixes, the model is connected to the input via an input layer like the following:

```
+ lstm_out = LSTM(128, input_shape=(maxlen, len(chars)))(net_input)
```

Transfer learning Transfer learning is a popular technique that takes an already-trained network with a different dataset. Then, the new model modifies the last layers to support the goal of the new problem and then performs some retraining without modifying the weights and biases of the layers from the previous network. We have observed several network connection fixes needed when the developer is attempting transfer learning. Generally these fixes change the last few layers of the DNN. One such kind of fix is shown below from *Stack Overflow* post #57248121[6]:

```
+ model_final.fit_generator(train_generator.flow(np.array(X_train), np.array(
    y_train), batch_size=32),
+ validation_data=test_generator.flow(np.array(X_test), np.array(y_test),
    batch_size=32),
+ steps_per_epoch=len(X_train)/32,
+ validation_steps=len(X_test)/32,
+ epochs=50)
```

In this example, the developer wants to train the imagenet with a pretrained network `VGG19` that has been used for face recognization. In this bug, the developer does not provide the correct data input size that leads to an error and fix was to include a data generator that loads the training data as expected by the `VGG19` model.

5.3.5 Add Layer

Finding 28 $\Rightarrow$ 30% of the add layers related fixes in *Stack Overflow* includes adding `Dropout` layer that can increase the training time ∼2-3x.

[6] https://stackoverflow.com/questions/57248121/

In a DNN model, adding layers helps to increase the performance and learn the features more accurately. We have found that a vast majority (∼30%) of these bug fix patterns includes the addition of the dropout layer. Dropout layer helps in removing the effect of the overfitting [122] that can also be achieved by using backpropagation. According to [122], backpropagation works better for the training dataset but does not work for new data. Dropout layer randomizes the structure of the architecture that helps the neural network to learn more features with every iteration. Dropout layer removes the connection between layers randomly stopping the data flow through those nodes and edges. Randomly reducing connections can have a negative impact on training time. With Dropout layers, the convergence of the training takes ∼2-3x more time [122].

5.3.6 Loss Function

Finding 29 ⇒ Among DNN hyperparameters, change of loss function happens to fix 6.2% (highest) of the bugs in *Stack Overflow* and 3.7% in *Github* that helps to enhance prediction accuracy and increase the robustness against adversarial attacks.

Loss function plays a crucial role in the convergence of the training process and in getting better accuracy during prediction. A model with wrong loss function does not learn the decision boundary of the features well and there can be overlap between the decision boundaries [8, 123] in the high dimensional feature space making the model vulnerable to adversarial attacks [103]. By a careful and deeper analysis of these loss function related fixes, we have found that they can be categorized into the following kinds:

Add new loss function. The fixes in this category involve adding a custom or built-in loss function. 10 out of 23 fixes fall into this category. In some of the fixes, it is needed to modify the network connectivity for the new loss function to work. For example, in the following fix of the bug in *Stack Overflow* post #51257037[7], the last layer is kept outside the gradient descent computation during training by adding `trainable = False`.

```
output = Dense(1, trainable = False)(hidden_a)
```

[7] https://stackoverflow.com/questions/51257037/

The custom loss function was designed by the developer in such a way that all but the output layer participate to lead to the convergence of the model. However, the convergence was not successful, as the output layer was actively participating in the forward and backward propagation that caused an abrupt change in the value of the loss function. Fixing these bugs require live trainable parameter analysis. This approach will help to monitor the active trainable parameters during the training to localize and fix these bugs. Currently, the developer needs to rely on theoretical knowledge to fix these bugs due to the lack of such kind of analysis frameworks.

Change loss function. 9 instances of bug fixes fall into the category of changing the loss function. Our analysis of these fixes reveals that the choice of these loss functions is sometimes confusing. Developers need to understand the data properties and the goal of the DNN task to come up with a proper loss function. For example, in constructing DNN models for classification problems, the developers are confused between the choice of `binary_crossentropy` and `categorical_crossentropy` as discussed in the fix of *Stack Overflow* post #45799474[8] and *Stack Overflow* post #42081257[9]. The first loss function works better for the binary classification problems; however, when the classification problem has more than two categories, one should use `categorical_crossentropy` as a loss function to avoid poor performance. Sometimes, the fix involves adding some filter to the mathematical operation used in the loss function. For example, we see the following bug fix of *Stack Overflow* post #34223315[10]

```
cross_entropy = -tf.reduce_sum(y_ * tf.log(tf.clip_by_value(y_conv, 1e-10,1.0)))
```

caused by the following line:

```
cross_entropy = -tf.reduce_sum(y_ * tf.log(y_conv))
```

In the above code snippet, the problem is that the user will get `NaN` values if `y_conv` becomes negative as the `log` of negative numbers is undefined. The fix adds a clipper function to filter out negative values to the `log` operation. In another fix of the same kind of bug in *Stack Overflow* post #42521400[11], `softmax` is used as the filtering operation that stops propagating values $<= 0$ to the `log` operation.

[8] https://stackoverflow.com/questions/45799474/
[9] https://stackoverflow.com/questions/42081257
[10] https://stackoverflow.com/questions/34223315/
[11] https://stackoverflow.com/questions/42521400/

```
softmax = tf.nn.softmax(logits)
xent = -tf.reduce_sum(labels * tf.log(softmax), 1)
```

5.3.7 Commonality of Fix Patterns in *Stack Overflow* and *Github*

Finding 30 ⇒ The p-value is 0.83 between the bug fix pattern distributions of *Stack Overflow* and *Github* indicating commonality of bug fix patterns in *Stack Overflow* and *Github*.

We have conducted a t-test at 95% significance level to understand the distribution of bug fix patterns in *Stack Overflow* and *Github*. The null hypothesis is H_0 : *the distributions are the same*. The null hypothesis is to be rejected if the p-value is less than 5% or 0.05. Our computation shows that the p-value is very high (0.83). So, H_0 can not be rejected concluding that the distributions are similar. We also notice that though in some bug fix categories e.g., data dimension, layer dimension, and versioning, there is a significant difference among the *Stack Overflow* and *Github* distributions, the other categories have a similar share of occurrences in *Stack Overflow* and *Github*. This indicates that the bug fix patterns have commonality across *Stack Overflow* and *Github*.

5.4 Fix patterns across bug types

To answer RQ2, we analyze the correlation between the bug types in the bug dataset presented by [3] and the bug fix patterns studied by this thesis using the distribution of the bugs and their corresponding fixes. The distribution of bug fix patterns across different bug types in *Stack Overflow* and *Github* are shown in the Figure 5.2 and 5.3, respectively. The horizontal and the vertical axes describe the different bug types from [3] and the percentage of different fix patterns needed to fix those bugs, respectively.

Finding 31 ⇒ For API bugs, fixing of the specifications between APIs is dominant (42% in *Stack Overflow* and 48% in *Github*) .

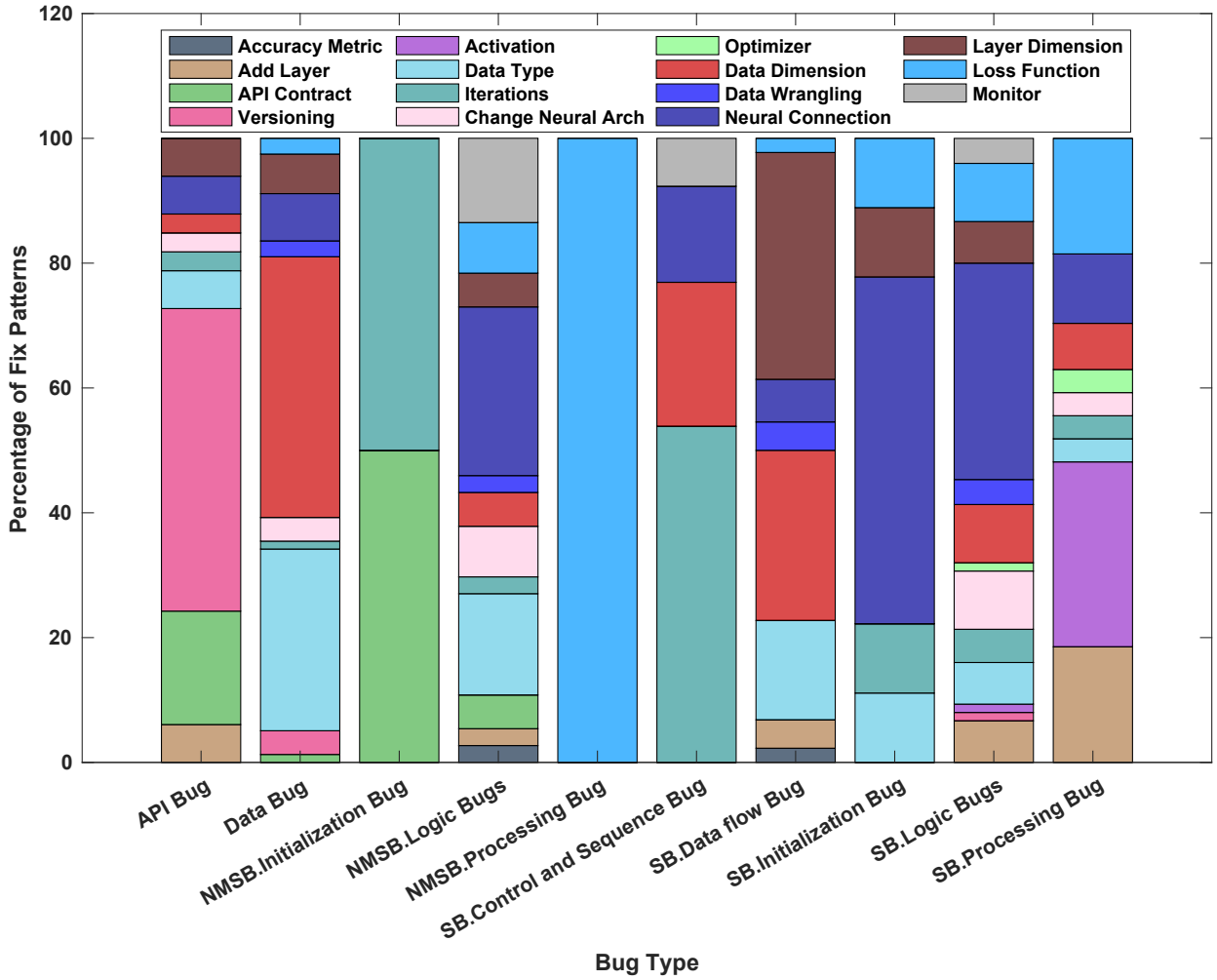


Figure 5.2: Distribution of Bug Fix Patterns for Different Bug Types *Stack Overflow*

Fixing API specifications involves changing API contracts due to API versioning and supporting inter-library operations within a model. Fixing API specifications is needed due to the following reasons:

Change of specifications due to version upgrade. 20 fixes in *Stack Overflow* involve changing specifications which are required due to the change of the library version. The changes during the upgrade of the library version involves the following changes: change fully qualified method names, change API signature, and change probabilistic behavior of the APIs. Though fixes due to the change of fully qualified method names and change of API signature are well-studied problems [124–126], the fixes due to the change of probabilistic behavior of the APIs are hard and different from traditional API changes. Localizing of these

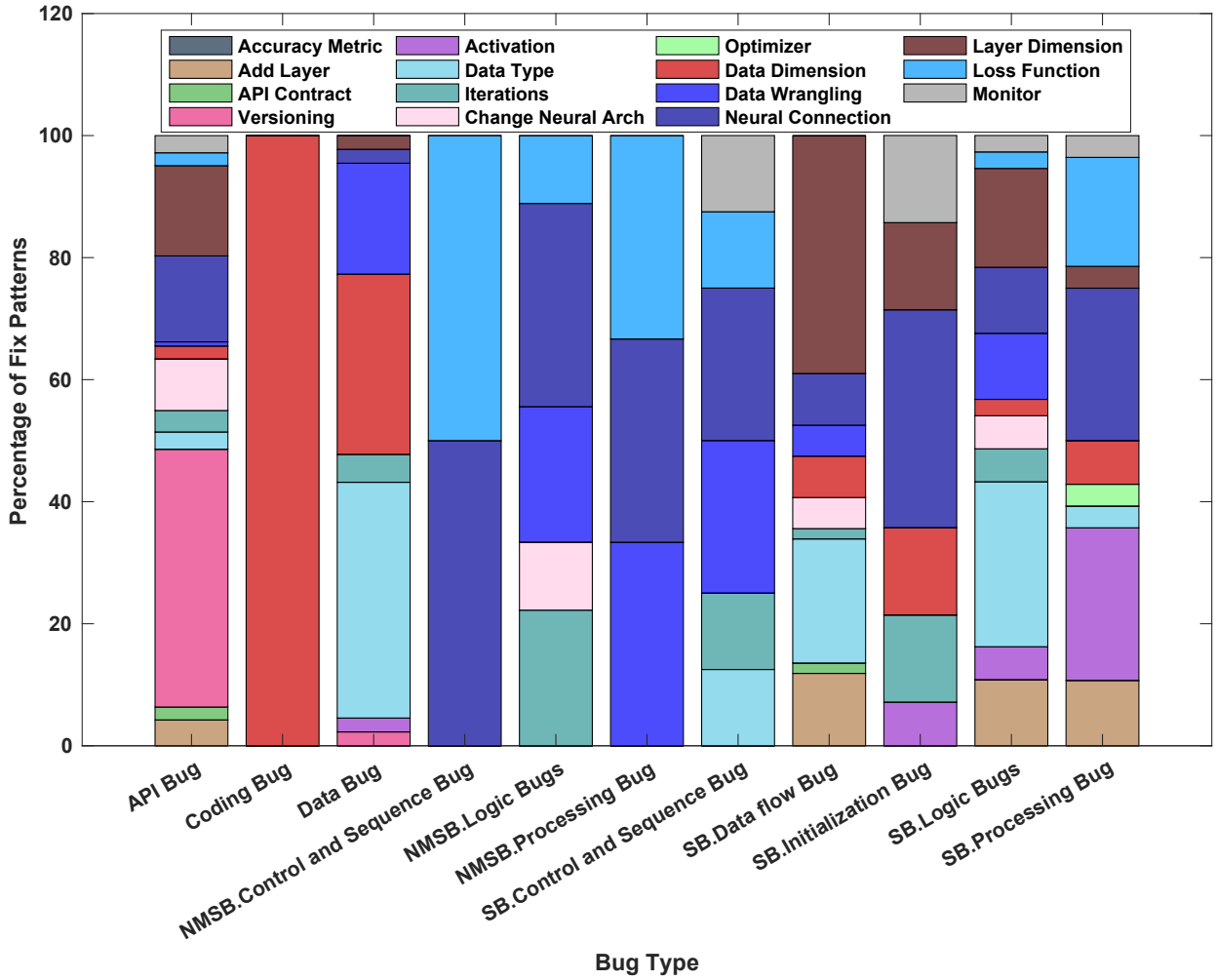


Figure 5.3: Distribution of Bug Fix Patterns for Different Bug Types *Github*

bugs are difficult due to the lack of sophisticated probabilistic analysis tools for DNN. For example, the bug discussed in *Stack Overflow* #49742061[12] says that the results are different in two versions of *Tensorflow*. The fix of this bug involves adding a dead code line that tweaks around the underlying probabilistic behavior of the APIs by overriding the modified random seed. The fix of *Stack Overflow* #49742061 is shown in Figure 5.4. The fix adds the line `Xf = tf.contrib.layers.flatten( X )` before the line `R = tf.random_uniform( shape = () )`. This addition overrides the random seed in the new version with the one in the previous version.

[12]https://stackoverflow.com/questions/49742061/

```
import numpy as np
import tensorflow as tf

tf.reset_default_graph()
tf.set_random_seed(1)
with tf.Session() as sess:
    X = tf.constant( [ [ 1, 2, 3, 4, 5, 6 ] ], tf.float32 )
    #Xf = tf.contrib.layers.flatten( X )
    R = tf.random_uniform( shape = () )
    R_V = sess.run( R )
print( R_V )
```

If you run this code as above, you get a printout of:

```
0.38538742
```

for both versions. If you uncomment the Xf line, you get

```
0.013653636
```

and

```
0.6033112
```

Figure 5.4: Fix of *Stack Overflow* #49742061

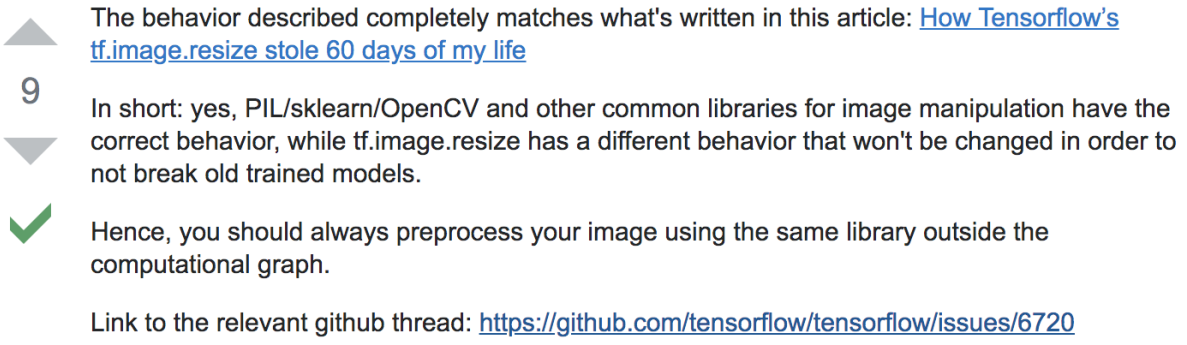


Figure 5.5: Fix of *Stack Overflow* #54497130

Our observation gives the intuition that the fix of versioning bugs due to the change of the probabilistic distribution in different version needs new DNN specific probabilistic analysis techniques.

Change specification to support interlibrary. In these fixes, the DNN program uses more than one library. These bugs arise due to the similar assumption of the behavior and specifications for different APIs in different libraries. Fixing of these bugs requires the expertise in both the libraries e.g., the bug discussed in *Stack Overflow* #54497130[13] that is shown in Figure 5.5. The discussion points to an issue in the official *Tensorflow* repository. The solution suggested to avoid using APIs from other libraries to pre-process images. However, in similar scenarios, the use of specialized image processing libraries is recommended to get better performance.

From Figure 5.2 and 5.3, we have found that fixing the data dimension is the most prominent pattern (41.77%) for fixing data bugs in *Stack Overflow*. For fixing data bugs in *Github*, the most prominent fix patterns are the change of data type (38.64%) and data dimension (29.55%). This suggests that for fixing data bugs, the major changes are related to data dimensions. This happens because the dimension of the data is very important for the correct functionality of the DNN model.

For fixing logic bugs the most common practice is to modify the network connectivity ($\sim$27.03% in *Stack Overflow* and $\sim$33.33% in *Github*). A detailed discussion on network connectivity is presented in §5.3.4. Whereas, a significant amount of data flow bugs can be fixed by changing the layer dimension ($\sim$36.36% in *Stack Overflow* and $\sim$38.98% in *Github*). A detailed discussion on fixing layer dimension is presented in §5.3.2.

These observations give us the intuition that for fixing different types of bugs, unique technical approaches might be needed.

5.5 Fix Patterns across libraries

To answer RQ3, we have studied the distribution of fix patterns across the 5 libraries. Then, we have conducted statistical pairwise t-test at 95% significance level between the libraries. Table 5.4 shows the p-values found from this test across the libraries.

We assume the null hypothesis is H_0*: the distribution of the fix patterns across two libraries are same.* If the p-value is less than 5% or 0.05, then we reject H_0. The p-value for the library pairs *Caffe-Theano*

[13]https://stackoverflow.com/questions/54497130/

Table 5.4: P-value of the distribution of Bugs btween the libraries

Library	*Caffe*	*Keras*	*Tensorflow*	*Theano*	*Torch*
Caffe	1.0	0.0045	0.00735	0.19	0.30
Keras	0.0045	1.0	0.84	0.0021	0.0024
Tensorflow	0.0073	0.84	1.0	0.0039	0.0044
Theano	0.19	0.0021	0.0039	1.0	0.80
Torch	0.30	0.0024	0.0044	0.80	1.0

(.19), *Caffe-Torch* (.30), *Keras-Tensorflow* (0.84), *Theano-Torch* (0.8) are much greater than 5%. So in these, cases we can not reject the null hypothesis. So, the libraries *Caffe*, *Theano*, and *Torch* show similar kind of bug fix patterns. The pair *Keras-Tensorflow* form a very strong related group with a p-value close to 100%. This suggests that similar kinds of automatic bug fix tools may be reused for *Caffe*, *Theano*, and *Torch* after converting into a common intermediate representation. Similarly, *Keras* and *Tensorflow* bugs can be fixed using similar technical approaches.

5.6 Introduction of Bugs Through Fixes

Finding 32 $\Rightarrow$ 29% of the bug fixes introduce new bugs in the code adding technical debt [31] and maintenance costs.

To answer RQ4, we have analyzed 100 randomly chosen fixes from *Stack Overflow* to understand whether fixing a bug can introduce a new bug. We have read the replies to the answers selected by filtering criteria discussed in §5.2. Then, we have identified whether the fix introduced new bugs by analyzing all replies to the answer fixing the original bug and classify them into bug type, root cause, and impact using the classification scheme proposed by the prior work [3]. We have found that 29% fixes in the randomly sampled dataset introduce at least one new bug in the code. Here, a new bug indicates that the original bug was fixed by the solution posted; however, the solution introduces a new bug that is different from the original bug type. Furthermore, we have compared the bug type, root cause, and the effect of the bugs of *Stack Overflow* posts with the newly introduced bugs and have found that only 6.8%, 13.8%, and 24.1% of the bugs match the classification of the parent bug type, root cause, and impact, respectively. This result shows that a

Table 5.5: Statistics of the Introduction of New Bugs During Bug Fix

Library	Bug Type						Root Cause						Impact		
	API Bug	Data Bug	SB.DF	SB.I	SB.L	SB.P	ATC	APIC	APIM	IPS	SI	UT	Bad performance	Crash	IF
Caffe	0%	0%	0%	0%	100.0%	0%	0%	0%	0%	33.3%	66.7%	0%	66.7%	0%	33.3%
Keras	30.8%	7.69%	30.7%	0%	23.1%	7.69%	7.69%	30.8%	0%	15.4%	15.4%	30.8%	23.1%	61.5%	15.4%
Tensorflow	60.0%	20.0%	0%	10%	10.0%	0%	20%	60%	0%	10%	10%	0%	20%	70%	10%
Theano	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	100%	0%	100%	0%	0%
Torch	50.0%	0%	50%	0%	0%	0%	0%	0%	50%	0%	0%	50%	0%	50%	50%

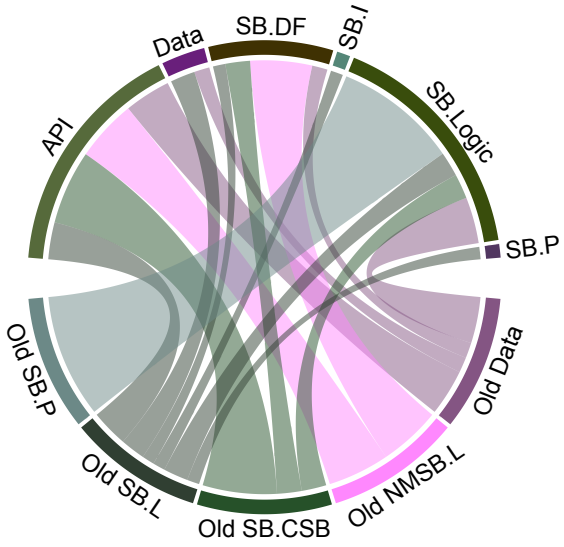
majority of the bugs introduced are of new types and their behavior is entirely different than that of the parent bugs'. In the Table 5.5, we have shown the distribution of the new bugs across the different libraries and how these new bugs are classified into different categories of bug type, root cause, and impact. We have also found that the *Crash* (55.8%) is the most common impact of these new bugs and a majority of these bugs are of *API Bug* (37.9%), and the most common root cause of these bugs are *API Change* (34.5%) that includes the change of either the signature of the API or the fully qualified name of the API. 44.8% and 34.5% of the newly introduced bugs are from *Keras* and *Tensorflow*. *Caffe*, *Theano*, and *Torch* related bug fixes introduce 10.34%, 3.45%, and 6.90% new bugs, respectively.

Finding 33 ⇒ 37.9% of the new bugs are from API Bug, 34.5% of them are due to API Change, and 55.2% of them end in a crash.

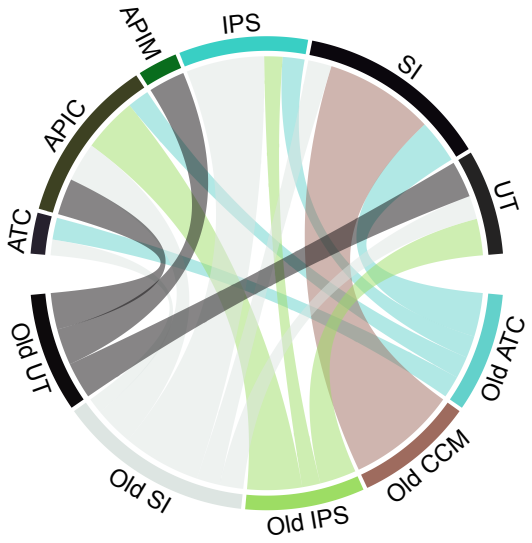
In Figure 5.6, the relation between the parent bugs' root cause, type and effect with the newly introduced bugs' distribution has been visualized. In this visualization, the *old* represents the parent bug and the relation has been drawn by a connection between two bug distributions. The width of the connection determines the strength of the relation. The perimeter covered by each bug type/root cause/impact depicts its overall strength. We have found that a large section of bug fixes introduces API bug and the major reason for that is the API change that mostly due to the versioning of the APIs and these fixes primarily lead to a crash and bad performance.

5.7 Challenges in Fixing Bugs

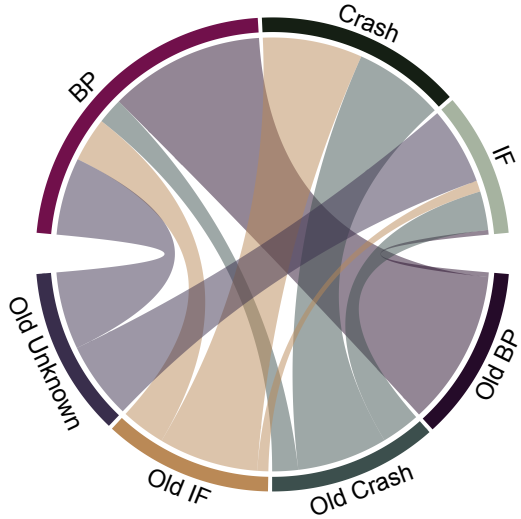
In this section, we explore the answer to RQ5 to identify the common challenges faced by the developers in fixing the DNN bugs. To understand the challenges, we have used a classification scheme separate from



(a) Root Cause



(b) Bug Type



(c) Impact

Figure 5.6: Bug fix pattern distribution: SB.P→SB.Processing, SB.L→SB.Logic, DF→Data Flow,SB.I→SB.Initialization, ATC→Absence of Type Checking, BP→Bad Performance, IF→Incorrect Functionality

bug fix patterns. Similar to the labeling performed for bug fix patterns, two raters have independently classified each post used in this study. These classes of new challenges are described below:

5.7.1 DNN Reuse

Training DNN models can be expensive because it requires sophisticated computational resources and a large amount of labeled data that might not be readily available. This has led to the reuse of DNN models that creates unique issues such as backdoor attack [127], injection of bias [128], and mismatch between the intent of the pretrained DNN model and the intent of the developer.

```
base_model = ResNet50(input_shape=(224, 224, 3),
include_top=False,weights='imagenet',pooling='avg')
+ x=base_model.output
+ x = Dense(512, activation='relu')(x) #add new layer
+ x = Dropout(0.5)(x) #add new layer
+ x = Dense(512, activation='relu')(x) #add new layer
+ x = Dropout(0.5)(x)
```

In the example above from *Stack Overflow* post # 49226447[14], the developer wants to train a predefined DNN model structure `ResNet50` using the cancer dataset. The trained network results in overfitting as the developer was not aware of the structure of the reused model and needed to modify the network by adding dropout and dense layers to reduce the effect of overfitting.

5.7.2 Untraceable or Semi-Traceable Error

In case of a crash bug, the common strategy to localize the bug is to analyze the error message. However, we have found that bug localization is very challenging in DNN software because errors and faults are non-trivially related. To illustrate, consider the code snippet below from *Stack Overflow* post # 33474424[15]:

```
model = Sequential()
model.add(Dense(hidden_size, input_dim=input_size, init='uniform'))
model.add(Activation('tanh'))
```

[14]https://stackoverflow.com/questions/49226447
[15]https://stackoverflow.com/questions/33474424

```
...
y_pred = model.predict(X_nn)
```

This code produces the following error trace:

```
ttributeError                            Traceback (most recent call last)
<ipython-input-17-e6d32bc0d547> in <module>()
1
----> 2 y_pred = model.predict(X_nn)
491     def predict(self, X, batch_size=128, verbose=0):
492         X = standardize_X(X)
--> 493         return self._predict_loop(self._predict, X, batch_size,
verbose)[0]
494
495     def predict_proba(self, X, batch_size=128, verbose=1):

AttributeError: 'Sequential' object has no attribute '_predict'
```

From this error message, a developer might start digging into the code of `predict` function and the `Sequential` object; however, the issue is the missing compilation step. Due to this, the model connection is not initialized and error propagates to the `predict` operation and halts the training process. We have studied randomly 50 bugs yielding `Crash` from *Stack Overflow*. **We have found that 11 out of 50 posts does not indicate any error message and in rest of the 39, 20 posts have a fix that does not match with the error message.**

5.7.3 Fast and Furious Releases

We have previously discussed that a large number of fixes are due to the rapid versioning and invasive changes in DNN libraries.

To study this challenge, we have labeled all removed, reconfigured, or renamed operations of *Tensorflow* from version 1.10 to 2.0 (latest in June 2019).

In Table 5.6, we have shown the number of symbols of operations available for each *Tensorflow* releases and the number of operations that have been deleted, renamed, or reconfigured in comparison to the previous

Table 5.6: *Tensorflow* API changes. Change= # of operations changed in comparison to the previous version.

Version	# of Symbols	Change	Release Date [129]
v2.0 (Beta)	6504	2185	Jun 7, 2019
v1.14.0	8363	59	Jun 18, 2019
v1.13.1	3560	39	Feb 25, 2019
v1.12.0	3314	52	Nov 1, 2018
v1.11.0	3145	175	Sep 25, 2018
v1.10.0	3230	N/A	Aug 7, 2018

version. **We have found that from the v1.14 to v2.0 26% of the operations have been changed**. We have also studied *Keras* v2.0, v2.1, v2.2, and v2.3 to understand whether this problem is only prevalent in *Tensorflow* or not. Our study has found that during the transition from v2.0-v2.1, v2.1-v.2.2, and v2.2-v2.3, the percentage of changes in operation are 6%, 8%, and 4%, respectively.

A non-trivial challenge for repairing DNN software is the probabilistic behavior of the APIs. Some of these version upgrades also change the probabilistic behavior of the APIs causing some difficult bugs. An example is presented below where the change of the probabilistic distribution changes the output of the same operation with different versions[16].

```
with Tensorflow 1.3
Z3 = [[-0.44670227 ...  0.46852064]
[-0.17601591 ...  0.5747785 ]]
with Tensorflow 1.4+
Z3 = [[ 1.44169843 ...  1.36546707]
[ 1.40708458 ...  1.26248586]]
```

5.8 Threats to Validity

External Threat. A source of external threat can be the dataset used to study the bug repair pattern. To alleviate this threat we use a benchmark dataset of DNN bugs prepared by [3].

Internal Threat. An internal threat can be the coding approach used to classify the bug fix patterns. We use the widely adopted open coding approach to come with a classification scheme to minimize this

[16]https://stackoverflow.com/questions/49742061

threat. Two Ph.D. students independently came up with the classification schemes. Then, these schemes were merged through moderated discussions. The expertise of the raters can be another source of an internal threat. We alleviate this threat by involving raters who have expertise in both the DNN libraries and the bug fix patterns. The raters were also trained on the coding scheme before the labeling. We also use kappa coefficient to measure the inter-rater agreement throughout the labeling process. And the value of kappa coefficient indicates that the labeling was successful with a perfect agreement.

Another threat is the number of posts in *Stack Overflow* for each library are not the same. To mitigate this threat, we have performed an ANOVA test on the *Stack Overflow* bug fix patterns. We have found that F (0.0002) < F-critical (2.50) that implies that the distribution of the bug fix in *Stack Overflow* is not unbalanced.

5.9 Discussion

We have analyzed the bug fix patterns in DNN and have found that there are significantly different new patterns compared to the non-ML bug fix patterns. There are also new challenges in fixing these bugs. In the analyses of RQ1, we have found that major fixes are related to data dimension, layer dimension, network connection, addition of layer, and loss function. To fix such issues, we need to know the structure of the network, how data flowing through the network is modified through various mathematical operations, how the performance evolves during forward and backward propagation due to the use of loss function, accuracy metrics, etc. This presents a number of immediate research challenges related to DNN API design and verification. To apply the data-related fixes, we need to understand the implicit dependencies between the data and model. This problem is akin to the notion of implicit coupling between modules. Studying various existing techniques to address strongly coupled data could be beneficial to fix the data-related problems. To fix the connections among consecutive layers in a neural network, the DNN model needs to be converted to a suitable common intermediate representation (IR). Then, we need to perform a reachability analysis to find the portion of the graph disconnected from the rest to fix such connection-related problems. Also, the fixes related to the addition of layer and change of loss function can be addressed automatically by mining specifications related to such layers and loss function from large codebases [130, 131].

In RQ1, we have also shown that some of the bug fixes have a negative impact on the robustness of DNN models [132]. Studying such cases further and developing tips for new developers is necessary so that they avoid falling into these traps without this knowledge. In RQ2, we have seen that bug fixes are different for different bug types. We have noticed that fixing API bugs require fixing the specification between APIs. These fixes can be achieved by validating the compatibility among APIs by adding robust test suites before releasing new versions. In RQ3, we have identified the commonalities among the fix patterns of different libraries. Efforts on repairing bugs in DNNs are certainly needed, and they can focus on these commonalities to cover more ground quickly. In RQ4, we have observed that fixing bugs can lead to new bugs. Our findings identifies some common situations where this happens, e.g., fixing layer dimension has the possibility of adding data bugs. We concluded our analyses by showing that new challenges are present in fixing DNN bugs. Though some of these fixing strategies have been adopted by existing tools, more work on validation and repair is warranted in several SE sub-communities such as program analysis, runtime verification, formal methods, etc. Analysis representation specific to DNNs can be developed to enable repair work. Runtime monitoring framework for DNNs would be useful to prevent errors from occurring and to collect traces for dynamic analyses based repair techniques. Safety critical data science applications of DNNs need these efforts to become more dependable [133].

5.10 Conclusion

The widespread adoption of deep neural networks in software systems has fueled the need for software engineering practices specific to DNNs. Previous work has shown that like traditional software, DNN software is prone to bugs albeit with very different characteristics. It is important to further understand the characteristics of bug fixes to inform strategies for repairing DNN software that has these bugs. How do developers go about fixing these bugs? What challenges should automated repair tools address? To that end, we conducted a comprehensive study to understand how bugs are fixed in the DNN software. Our study has led to several findings. First of all, we find that bug fix patterns in DNN are significantly different from traditional bug fix patterns. Second, our results show that fixing the incompatibility between the data and the DNN alone can be of significant help to developers of DNN software, especially if the developers can

be warned about the impact of their bug fix on the robustness of the DNN model. Third, our study shows that a prevalent bug fix pattern is version upgrade. While version upgrade is well-studied in SE research, our bug fix patterns suggest that automated repair tools will need to address at least two unique challenges: invasive, backward incompatible changes and probabilistic behavior change. Fourth, our study shows that the structure of the DNN itself needs to be represented in repair tools because several fix patterns rely on identifying incompatibilities in that structure. For instance, network connection fixes where disconnected layers are identified and connected, or adding missing layers, etc. Fifth, we have found that a significant number of bug fixes introduce new bugs in the code. Finally, we have identified three challenges for fixing bugs: bug localization is very difficult, reuse of the DNN model is hard because of limited insights into its behavior, and keeping up with rapid releases is hard.

This study opens up several avenues for future work. First and perhaps most immediately, a number of bug fix patterns identified by this work can be automated in repair tools. Such tools for bug repairs can help the developers integrating DNN into their software. Second, an abstract representation of the DNN along with the code that uses it can be developed. We saw several bug fix patterns that rely on analyzing such a representation. Third, there is a critical need to improve bug localization for DNN by addressing unique challenges that arise, and by creating DNN-aware bug localization tools. Fourth, there is an urgent need to detect bugs introduced by dimension mismatch and specially changes that have the potential to introduce vulnerabilities in the DNNs. Fifth, urgent work is needed on upgrade tools that encode the semantics of version changes and keep up with the change in the signature and semantics of DNN libraries. This is important to keep pace with rapid development in this area.

CHAPTER 6. Amimla: MISUSE DETECTION FOR MACHINE LEARNING APIS

6.1 Introduction

Libraries and frameworks have been developed to facilitate the rapid increase in the popularity of machine learning (ML) technologies. Their functionality via APIs provides the building blocks that abstract away the complex details behind the stages in ML pipelines, such as data preparation, model creation, training, evaluation, tuning, and prediction. Using ML libraries/frameworks eases the tasks for ML researchers and practitioners.

By design, the goal of the libraries is also to make the code using their APIs reliable. However, the abstraction in the libraries' APIs might also introduce constraints on the components of a single API or among multiple APIs in order to use the libraries correctly. For example, method `keras.models.Model.predict()` expects the first argument to be a `numpy array`. In another example, in *Keras*, if `backend.set_image_dim_ordering()` is called with argument value of `'th'`, which means *Theano* is used in the back-end, then the `input_shape` of any `Convolution2D` layers constructed later must be `(batch, channels, rows, cols)` whereas if `backend.set_image_dim_ordering()` is called with argument value of `'tf'` to use *Tensorflow* in the back-end then the `input_shape` must be `(rows, cols, channels)`.

Violating constraints lead to **misuses** of the APIs and would result in crashes or poor performance [4, 134]. An example of single method misuses is discussed in *Stack Overflow* question 37891954 where the user passed a `list` instead of a `numpy array` as an argument to `keras.models.Model.predict()`. An example misuse with multiple methods is discussed in *Stack Overflow* question 41771965 where the `input_shape` of `(channels, rows, cols)` was used with *Theano* in the back-end.

There has been a rich body of work on studies and detection for general API misuses as described in [58]. However, there have not been any such studies or detectors designed for ML APIs. Closest related works are by Zhang *et al.* [51], Thung *et al.* [52], and Islam *et al.* [4, 134]. Zhang *et al.* [51] have analyzed

bugs in software that make use of the *Tensorflow* library, Thung *et al.* [52] have analyzed bugs in the code of three ML systems. Islam *et al.* have analyzed bugs [134] and fixes [4] in code that uses five deep learning libraries *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch*. In this chapter, we conducted an empirical study on the issues with using ML libraries discussed in more than 2,000 *Stack Overflow* posts. The result confirmed the existence of ML API misuses in practice with as much as 13% of the posts for *Keras*. We also observed several characteristics of ML misuses that are different from general APIs and are not supported by existing API misuse detectors. We observed that ML API usages, especially in deep learning, require specific constraints between layers in the network, e.g., compatibility between the dimensions of the output of a layer and the input of the next layer, which never exhibit in previous studies of misuses for general APIs. Manually checking this kind of constraint is not always possible because the concrete values of dimensions might only be available at runtime. We also observed that the majority of ML misuses in our study are related to arguments and values of arguments that are not supported by existing approaches.

The observations above motivated us to develop Amimla (Api Misuse In Machine Learning Apis), an automated approach specific to detecting ML API misuses. Amimla builds abstract ML pipeline for libraries and uses them to detect missing stage(s). Amimla extracts abstract deep neural networks (DNNs) from ML programs and performs symbolic analysis on abstract DNNs to detect dimension incompatibility misuses. Amimla creates a canonical form of APIs to detect type related misuses. Amimla also extracts constraints on arguments of single methods and constraints between multiple methods to detect misuses.

Our empirical evaluation shows that Amimla performs well on the misuse dataset from which we drew the observations as well as on an independent dataset extracted from *Github* ML projects. In both cases, Amimla achieved the precision of more than 70% and the recall of more than 80%. We also applied Amimla on the working versions of 26 real-world open source projects and detected 52 not-yet-found ML API-related bugs. We reported them to the respective projects, 3 of which have already been confirmed by the developers as of this writing.

In summary, this work makes the following contributions.

1. We present a *classification of the ML API misuses*.

Table 6.1: Classification of Machine Learning API Misuses.

			Data Prep.	Modeling	Training	Evaluation	Tuning	Prediction
Method Call	Single	Receiver			1|0			
		Method	2|0	9|3	4|3			1|0
		Argument	3|0	38|35	13|13	1|0	2|7	0|4
	Multiple	Method Order		2|0				
		Method Calls	2|1	13|8	8|3		1|1	0|1
		Argument		4|4	1|0		1|0	
Iterations			1|0		2|1			
Conditions					0|1			

In each cell, the pair of values respectively indicate the numbers of missuses for *Tensorflow* and *Keras* found in our study.

2. We have also created datasets of misuses from *Stack Overflow* and *Github* that could be utilized by other researchers conducting similar studies.

3. We have created good usage patterns for *Keras* and *Tensorflow* APIs using trusted applications from *Github*.

4. We have developed Amimla a framework to detect the misuses in deep learning applications using popular libraries *Keras* and *Tensorflow*.

5. Finally, we have applied Amimla to 26 real-world open source projects, and detected and reported 52 not-yet-found ML API-related bugs.

The rest of this chapter is organized as follows. Next, we present our empirical study that has used *Stack Overflow* posts to understand the characteristics of ML API misuses. Section 6.3 presents our API misuse detection approach. Section 6.4 describes the evaluation of our approach, Section **??** presents related ideas, and Section 6.5 concludes.

6.2 Motivation: Study of ML API Misuses

6.2.1 Data Collection

We collected data from *Stack Overflow* posts to study ML API misuses. *Stack Overflow* is a well-known Q&A site for developers discussing programming problems. The data collection process consists of two

steps: 1) we used several criteria, discussed below, to select candidate posts discussing the use of ML APIs and 2) we manually read these candidates to identify the ones about API misuses. To focus on the high-quality posts and keep the manual effort manageable, we selected the posts whose scores were greater than 5. The score of a post is a commonly used quality indicator [135]. It is computed as the difference between the number of its upvotes and the number of its downvotes.

In the first step, we focus on *Tensorflow* and *Keras*, which are the two most discussed ML libraries on *Stack Overflow*. When posts are about specific libraries, they are more likely to talk about APIs. Using these criteria, we selected 1,941 and 641 posts about *Tensorflow* and *Keras*, respectively. We did that by searching for posts that were tagged with *Tensorflow* or *Keras*. We further filtered the posts that did not contain any source code because posts about API misuses usually contain code snippets. After this step, in total, we selected 1,285 posts for *Tensorflow* and 430 posts for *Keras*. By post we mean the questions with unique id. However, for manual analysis we also study the answer for the questions,

In the second step, two Ph.D. students with expertise in machine learning and familiar with these libraries manually reviewed the candidates. For each post, they read the question and all answers focusing on the best-accepted one. If the best-accepted answer was to fix the usages of the ML API(s) in the question, they considered that post as talking about ML API misuse. Cohen's kappa coefficient [109] of agreement between the reviewers is 92% indicating a perfect agreement. After this step, we selected 109 and 85 misuse posts for *Tensorflow* and *Keras*, respectively, where both reviewers had an agreement. The reviewers were also asked to classify the misuses using our ML API misuse classification which is described next.

6.2.2 ML API Misuse Classification

Our classification takes two orthogonal views on the misuses. First, we look at the elements of the APIs or the roles of the API elements involved in the misuses. This point of view is similar to the criteria for API misuse classification in MuC [58]. For example, we checked if a misuse was on the value of a single method call or a violation of the order between two methods, or there was a missing condition check or control structure in the API usage. Second, we also look at which stage in the ML pipeline a misuse happens. This

is important because, in ML libraries, APIs are specifically designed to provide the functionality to support different stages in ML pipeline, e.g., data preparation, modeling, training, evaluation, tuning, and prediction.

Table 6.1 shows our classification for ML API misuses. Rows are for API elements and columns are for ML stages.

6.2.2.1 Machine Learning Stages

A ML pipeline is typically divided into 6 stages: data preparation, modeling, training, evaluation, tuning, and prediction. *Data preparation* stage converts raw input data into clean input data suitable for fitting to ML models. *Modeling* stage includes selecting a suitable model and constructing the model using the APIs. *Training* stage takes the model and adjusts its weights using the input-output examples known as training data. *Evaluation* is the stage where the trained models are evaluated against the test data. *Tuning* is the stage where parameters such as learning rate is tuned to improve the performance of the trained models. *Prediction* is the final stage in the pipeline where the trained and tuned models are use to predict new data.

6.2.2.2 API elements

Since all ML API usages involve method calls, the top level of our classification contains method calls, and conditions and iterations controlling method calls.

Method calls are the basic blocks of APIs. A method call in *Python*, e.g., `r.m(arg=val)`, consists of an optional receiver object on which the method is called, e.g., `r` in this call, a method name, e.g., `m`, and a list of arguments which could be empty. An argument is a pair of `name=value` where the name, which could be omitted, must match the name of a formal parameter of the method. A misuse could happen to some component(s) of a **single method** or happen with **multiple methods** if it violates the constraint between them.

Misuses in **Iterations** category are for problems with the looping control structure in ML programs. A misuse could occur when a usage is executed only once while it should be in an iteration, i.e., *missing iteration*, or when a usage is executed more than once while it should be run only once, i.e., *redundant iteration*.

Misuses in **Conditions** involve the preconditions when calling API methods. A *missing conditions* misuse occurs when a usage does not follow the API usage constraints to ensure certain mandated conditions. A *redundant conditions* misuse occurs when there is a condition which restricts a compulsory part of a usage.

6.2.3 Observations

We have the following observations from the study.

O1. There are a significant number of questions about ML API misuses. 109 and 85 posts are about API misuses among 1,941 (6%) and 641 (13%) posts about *Tensorflow* and *Keras*, respectively. These high rates show that API misuses also exist in ML API usages.

Developers seem to struggle the most with using APIs in model selection and construction with 66 posts (61%) for *Tensorflow* and 50 posts (59%) for *Keras*, and also in training models with 29 posts (27%) for *Tensorflow* and 21 posts (25%) for *Keras*.

O2. Most ML API misuses cannot be detected by existing approaches. From Table 6.1, we can see that the majority of misuses are related to argument and argument values. Among 109 *Tensorflow* and 85 *Keras* misuses, 57 (52%) and 59 (70%) are related to argument of single method calls, and 38 (35%) and 49 (58%) are related to argument values for *Tensorflow* and *Keras*, respectively. To the best of our knowledge, none of the existing detectors can handle misuses in types and values of API arguments. The state of the art classification in MuC [58] also does not have argument types and values in its taxonomy.

O3. Network layer dependent misuse of ML APIs. Some API misuses occurred when the constraints on the properties of consecutive network layers were not respected. The following code snippet shows such an example misuse extracted from *Stack Overflow* question 34311586

```
model = Sequential()
act = PReLU(alpha_initializer='zero', weights=None)
model.add(Dense(64, input_shape = (128,), ...))
...
model.add(Reshape([32,1]))
...
model.add(Dense(32, ...))
model.add(Reshape([32,1]))
```

Line 5 causes the program to throw exception when adding a `Reshape` layer to the network after a `Dense` layer. The reason is that the usage did not respect the constraint between `Reshape` API and the previous `Dense` API that requires the product of the numbers in the list inside `Reshape`, 32×1 in this case, has to be equal to the value passed to the first argument, $arg0 = 64$, of the previous `Dense` layer.

These observations motivate us to develop a new approach to efficiently detect API misuses of ML libraries. The next section will describe our proposed approach in detail.

6.2.4 Threats to Validity

Internal Validity One possible threat would be the classification of the API misuses in our study. To alleviate this threat, two Ph.D. students were asked to label the misuses individually. We used Cohen's Kappa statistics [109] to measure the rate of agreement between the raters. Another possible internal threat would be the classification scheme used in Table 6.1. To alleviate the threat, we have adapted the classification from [105] and added some categories when needed following the well established open coding strategy [85–87].

The ML expertise of the raters can affect the manual labeling. To mitigate this threat we selected raters who have expertise in ML as well as using the libraries in the study. The raters also studied the answers and comments in posts to improve their insights.

External Validity In *Stack Overflow* threat to validity can be low-quality posts [92], and chronological order of posts. To eliminate the quality threat we studied only the posts that have the tag of the relevant library and then only kept the posts that have score ≥ 5 as suggested in [135]. This balanced both the quality and labeling efforts.

Chronological order of the posts can introduce threat as some older posts may be resolved in a later version of the libraries. To alleviate this threat the raters carefully observed the posts and if some post contains APIs that are outdated were discarded.

An external threat can be the expertise of the developers asking the questions. The observations gained by studying only the questions by new developers may not generalize. To understand this threat, we measured *reputation*, a metric used by *Stack Overflow* to estimate expertise. *Stack Overflow* users above 50 are

considered reputable and are allowed to comment on posts. The mean reputation of developers asking the questions about Keras and TensorFlow were 1375 and 1399 respectively indicating that the expertise of the developers asking the questions is not a threat to our study.

6.3 Amimla: ML API Misuse Detection

We now describe Amimla, our automated technique for ML API misuse detection, that builds on the observations described in Section 6.2. The key insights in our technical approach are following:

An abstract representation of DNN. A number of ML APIs provide support for deep neural networks (DNN). While treating methods in those APIs as black box is certainly possible, we would loose much of the semantics of the DNNs. On the other hand, if we expanded our analysis to include the body of the API methods it would quickly become non-scalable because of the complex numerical operations implemented within these API methods. To that end, our key insight was to create an abstract representation of the neural network that we call *abstract neural network* (ANN). ANN encodes relevant details of the DNNs as they are being constructed by the modeling APIs, but at the same time helps us avoid the scalability problems. We use ANNs to detect dimensional incompatibility mismatches between consecutive layers that may arise during the construction of the network. Since the dimension values might not always be concrete, e.g., involving the input parameters of the APIs, we perform symbolic analysis over ANNs during detection.

An abstract representation of ML pipeline. An important step in API misuse detection is to extract sequences of APIs from code. Then, these sequences are utilized for detecting misuses that are violations of temporal patterns, e.g. [70]. In addition to these misuses, in ML API usage we have found that ordering between categories of APIs need to be checked, e.g. training related APIs need to be called before prediction related APIs. To solve this problem, our insight was to create an abstract representation of ML pipelines, infer these representations from code, and then use the inferred representations to detect misuses where the temporal ordering of stages does not conform to the expected order. An abstract pipeline is a sequence of APIs performing the corresponding stages in an ML pipeline. It does not contain any computation in those stages.

Utilizing a canonical form. Another challenge in adoption of prior representation of API sequences to ML APIs arises from the nature of programming languages. While previous work has relied on a unique API signature because the order of actual arguments in a method call is fixed for languages like Java, common languages used for ML such as Python do not impose such constraints. This leads to a combinatorial explosion in the number of legal API sequences. To solve this problem, our insight was to convert each usage into a canonical form that reduces the number of legal API sequences.

Leveraging documentation and good usage. We also make use of good usage and documentation to infer constraints on the API usage that is then utilized to detect misuses.

Figure 6.1 shows an overview of our approach which can be summarized into the following steps:

1. First, we create an API canonical form of ML API calls. This helps to detect the type mismatches, incorrect argument keywords of the APIs. We collect good usage API canonical forms and store in a database to detect misuses in new programs by checking against the oracle.

2. Then, we create the fully qualified method names of the APIs from the documentation. We store this in a database to check the fully qualified method names of APIs in target programs.

3. Then, we study top-rated *Stack Overflow* posts to extract single-method parameter and multiple-method parameter constraints and store them in separate databases.

4. Then, we take a target program and create API sequence with the canonical form of the APIs, create abstract NN and abstract pipeline, and fully qualified names of the APIs.

5. Finally, we detect misuses using the intermediate representations and the pre-built databases.

6.3.1 Abstract Neural Network (ANN)

From a deep learning program, we extract the ANN as a graph. Each ANN node represents a layer API and each ANN edge represents the order between layers. An ANN node has two properties representing the corresponding layer's state: values of arguments and output dimension. To create an ANN of a program, we first parse the source code to create the abstract syntax tree (AST). We then visit the AST to create ANN

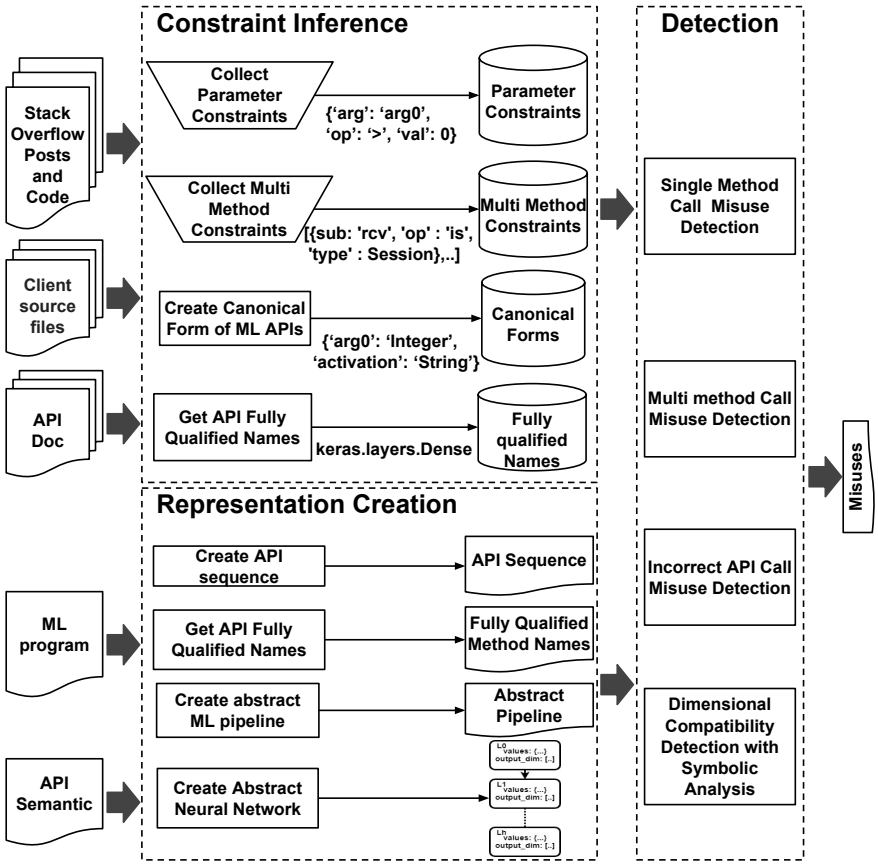


Figure 6.1: Overview of our detection approach and Amimla tool.

nodes and connect edges between nodes in the order of visiting. We rely on the semantics of layer APIs of the libraries and operations that add layers to the DNN to extract the ANN as a graph as shown in Figure 6.3 for the program shown in Figure 6.2. Each ANN node for a layer is annotated with the API name so that we can check whether an expected dimensional constraint is violated between the APIs. We extract the values of the parameters of the APIs from the AST. We compute the output dimension of the layer using symbolic analysis and the semantic formula of the APIs. As an example, let's see how the output dimension of layer 0 in Figure 6.3 using `Conv2D` API is computed.

From the argument values used in this layer we see that `kernel_size=(3,3)`, `strides=(1,1)`, `arg0=32` (`arg0` represents the number of filters used in the convolution operation) and `input_shape=(128,128,128)`.

```
1  img_width, img_height = 128, 128
2  model = Sequential()
3  model.add(Conv2D(32,kernel_size = (3,3),
     strides = (1,1), input_shape = (
     img_width, img_height, 3), ...))
5  model.add(Reshape([126*126, 32, 1]))
6  model.add(Activation('relu'))
7  model.add(MaxPooling2D(pool_size = (2, 2)
     , name = "pool1"))
8  model.add(Flatten())
9  model.add(Dense(10))
10 model.add(Activation('sigmoid'))
11
```

Figure 6.2: Code excerpt from Stack Overflow
Building abstract NN from source code.

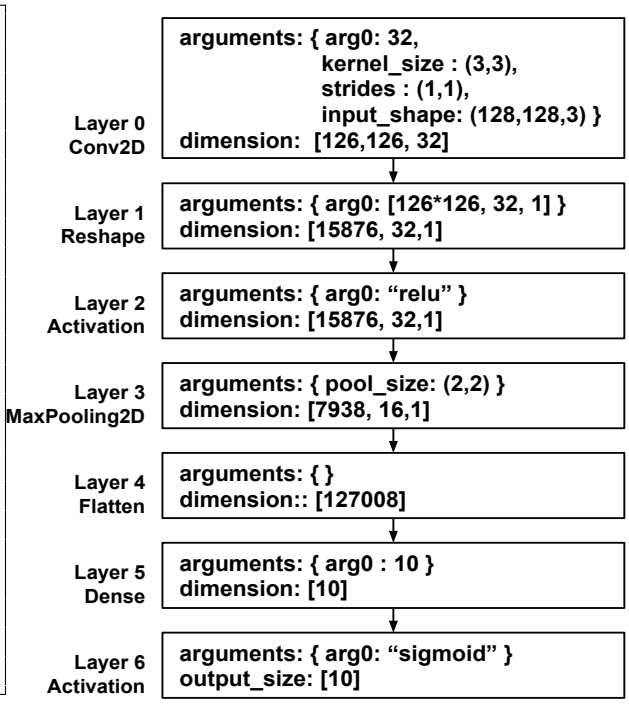


Figure 6.3: Abstract Neural Network

So we evaluate the formula through this layer to compute the output dimension. The formula to compute dimension due to convolution is following:

$$D' = (D - K + 2 \times P)/S + 1 \tag{6.1}$$

where D is the dimension passed to the convolution API and D' is the dimension produced by the `Conv2D` API, K is the kernel size, P is the padding size and S is the stride length. As the result, the output dimension due to this `Conv2D` will be

$$[(128 - 3 + 2 \times 0)/1 + 1, (128 - 3 + 2 \times 0)/1 + 1, arg0] = [126, 126, 32] \tag{6.2}$$

6.3.2 Abstract ML Pipeline

The ML pipeline contains the APIs used in the creation of machine learning stages. We get the information of different APIs used to construct the different stages from API documentation. Then, we extract the abstract ML pipeline by visiting the abstract syntax tree (AST) of the program. The abstract ML pipeline

is an API sequence. The sequence does not contain any information about actual computation performed during these stages. We use this API sequence of ML pipeline to identify if there are any missing stages in the pipeline. For example, the absence of `fit` API in the following program indicates the missing of the training stage which may cause the whole machine learning process to fail.

```
...
model = Sequential()
model.add(Dense(32, "relu"))
model.add(Dense(10, "sigmoid"))
model.predict(X)
```

The program fails to predict successfully as the model has not been trained. The program does not show any exception or error which can help the developer to identify the mistake. The developer should either `load` a pre-trained model or `fit` the model with the data before being able to make predictions successfully.

We create an ideal pipeline for each of the libraries. To create the ideal pipeline with required and optional stages by each library, we collect the required and optional relation between the APIs used to create stages by studying *Stack Overflow* answers and discussions, extracting the pipeline from *Github* codes and also consulting the documentation. For example, as per *Keras* documentation, it is required to first `create` a model, then `compile` the model, then `fit` the model and then `predict` on the model. However, from *Stack Overflow* we have found that instead of using `fit` we can load a pre-trained model using `load` API. We have also collected pipeline as API call sequence while visiting the *Github* codes. We combine the three information and manually verify the final pipeline. An example of the abstract pipeline for a *Keras* model is represented in the following way:

```
[split, normalize, replace, reshape,shuffle,...],create, compile, {load, fit},[
    evaluate],[scan,...], predict
```

In the above code, ... means some APIs in that part is omitted here. `split, normalize, replace, reshape, shuffle,`

`create, compile, load, fit, evaluate, scan,`

`predict` are different APIs to build stages of a ML pipeline in *Keras*. For example, `[split, normalize, replace,`

`reshape,shuffle]` APIs are used in the data preparation stage, `create` is used in the model creation stage, `compile, fit` together are used in the training stage, `evaluate` is used in the evaluation stage, `scan` is used in the tuning stage and `predict` is used in the prediction stage. The multiple options inside `{...}` indicates any one of the option should be present in the pipeline. The options inside `[...]` indicate zero or more of these may be present in the program. A ML stage is considered required if it is found in all the error-free complete ML programs we mined. We mark the stage as optional if we find some stage is missing in at least one program but the program still runs correctly. By run correctly, we means the model training can still be performed successfully without any error during the training time. We store the ideal pipeline for each library for detection of misuse in new programs.

6.3.3 API Canonical Form

API canonical form is the representation of APIs in the form of objects of key-value pairs where keys are argument names and values are argument data types. We can directly load those objects as a *Python* dictionary to do a constant time check.

For example, the following call to the `Dense` API

```
Dense(32,'input_dim' = 10,'activation' = 'relu')
```

has the following API canonical form

```
{'arg0': 'Integer', 'input_dim': 'Integer', 'activation': 'String'}
```

The API canonical form is important for analyzing the ML APIs written in *Python* language as the keyword arguments in Python can be passed in any order while calling the APIs.

For example, the above `Dense` API can also be called in the following way by swapping the order of `activation` and `input_dim` keywords.

```
Dense(32, activation = "relu", "input_dim" = 10)
```

Using API canonical form, all permutations of arguments with the same semantics, which is normally large due to a large number of arguments in ML APIs, will be represented by the same form. The state-of-

the-art sequence representation in [70] would consider different permutations as different usages leading to incorrectly identifying some good usages as misuses.

Type resolution is an important challenge for creating API canonical form. If in some of the cases we can not extract the type, we will keep the type as `unknown`. We also use a def-used analysis for best-effort type resolution. For example, the following call of `Dense` API

```
Dense(units = 128, input_shape = (128, 32), name = "input")
```

we extract the API canonical form as

```
{'units': 'Integer', 'input_shape': {'arg0': 'Integer', 'arg1': 'Integer'}, 'name':
    'String'}
```

Now let's suppose we see a call of `Dense` API like following later in the control flow:

```
Dense(units = getUnits(),input_shape = (128,32),name = "input")
```

This call alone doesn't provide sufficient information to extract the type of `units` but from a previous usage, we have information that `units` is of `Integer` type. So we replace the type here with the previous known type.

In *Python*, if keyword argument is used in the call then we use that keyword as the key; otherwise we create names `arg`i where i is the corresponding index of the argument in the formal parameter list of the API.

6.3.4 Storing Good usage API Canonical form and abstract ML pipeline

For the collection of API canonical form and ideal abstract ML pipeline, we have used 200 top repositories for *Keras* and *Tensorflow* based on star count. We used the API names of *Keras*, *Tensorflow* as keywords to find the repositories contains *Keras* and *Tensorflow* programs. We also manually verified random 50 of these repositories to confirm that, these repositories contain the target ML programs. Then we extract the canonical form of the APIs as discussed in Section 6.3.3 and store them for detection. We also store the ideal abstract ML pipeline as discussed in Section 6.3.2.

6.3.5 Collecting Constraints from *Stack Overflow*

We study top voted *Stack Overflow* posts about *Keras* and *Tensorflow* and collect the constraints on APIs both for single API call parameters and multiple API calls through manual process. We find out as many constraints as we can through manually studying the *Stack Overflow* posts, answers and discussions taking inspiration from by [4, 51, 134, 136].

6.3.5.1 Single API Constraints

A **single method call parameter constraint** for an API is represented as a list. Each element of the list contains a set of relational expressions that have to hold all together. Each relational expression specifies a condition on an argument with three components: the argument name `arg`, the relational operator `op` and a value `val` in the right-hand side of the expression. For example, the `Dense` API has the following constraint `{('arg': 'arg0', 'op': '==' , 'val': 1), ('arg': 'activation', 'op': '!=','val': 'softmax')}`. This constraint means that if the first argument is 1 then the activation function should not be `softmax`. Then the following call `Dense(1, 'activation': 'softmax')` will violate that constraint which indicates a misuse of `Dense` API. The single API constraints are collected by a thorough study of *Stack Overflow* posts and answers. We collected 100+ constraints and stored using the above representation. The violation of these constraints can lead to either performance pitfalls or runtime exception. **A unique aspect of ML API misuse detection is that the violation of some constraint may not lead to runtime error or exceptions but can cause the training process to not converge.** There can also be constraints that throw exception at runtime. For example, in the following call `Dropout(-1)` will throw exception.

The constraint for the call of `Dense` is `{('arg': 'arg0', 'op': '>' , 'val': 0), ('arg': 'arg0', 'op': '<','val': 1)}`. So the argument value should be greater than 0. If the value is 0 then having Dropout does not have any effect. And if the value should also be less than 1. If it is 1 all the edges will be dropped out having no data flow in the following layers.

6.3.5.2 Multi-API Constraints

We collect the **multiple method constraints** from *Stack Overflow* and documentation of the APIs. The multiple method constraints specify the requirements on the temporal order between APIs in the usages. The requirement could be that certain API be called if some API has been called or certain parameters could be passed to the later API call if some API has been called or certain API could be not allowed if some other API has been called. We represent a multiple-method constraint in the form of a tuple `<action, call_sequence>`. The actions can be `require` or `disallow`. If `disallow` action is used then the API call order in the `call_sequence` is not allowed and is reported as misuse. If `require` action is used then the API call order in the `call_sequence` must be enforced.

To illustrate consider the following misuse of `run` API attempting to use a closed session.

```
sess = tf.Session()
sess.close()
sess.run(...)
```

The code snippet above shows an exception at runtime. For the `run` API, we have the following constraints:

```
1 <"require",  "tf.Session()", "Session.run()",  "Session.close()">
2 <"disallow", "Session.close()", "Session.run()">
```

All these constraints are violated for the `run` API in the above example. Violation of multi API constraints sometimes can also have performance pitfall. For example, the constraint `<"require", "shuffle()", "split()">` asserts that before splitting the data for training and testing we need to shuffle the data. Violation of these constraints will not lead to any exceptions but can lead to a bad performance at prediction. These constraints are then stored for the detection of violation in new programs.

6.3.6 Storing Fully Qualified Method Name of APIs

We collect the fully qualified names of the API methods from API documentation. We store them in a database so that we can check the new ML program against this oracle. From the new ML program, we extract the fully qualified method name using the import statements. In case of ambiguity, while extracting

```
30      -       cost_l2 = tf.nn.l2_loss(tf.sub(infer, rate_batch))
    30  +       cost_l2 = tf.nn.l2_loss(tf.subtract(infer, rate_batch))
31  31          penalty = tf.constant(reg, dtype=tf.float32, shape=[], name="l2")
32      -       cost = tf.add(cost_l2, tf.mul(regularizer, penalty))
    32  +       cost = tf.add(cost_l2, tf.multiply(regularizer, penalty))
```

Figure 6.4: Bugfix due to wrong fully qualified method names. Red color indicates the deleted code and green color indicates the added code.

the fully qualified name, we consider all the ambiguous fully qualified names as valid. If none of those names appear in the oracle then we mark that as a misuse.

```
from keras.layers import *
from keras.model import *
model = Sequential()
model.add(Dense(...))
```

For example, in the following program, when we extract the fully qualified method name of `Dense` API from the program both the names `keras.layers.Dense` and `keras.model.Dense` can be candidate fully qualified name. So we keep both of them and check whether any of them appears in the oracle. Keeping this information helps us to detect a number of misuses that we have found on *Stack Overflow* and *Github*. For example, the code snippet from *Github* in Figure 6.4 contains misuse due to the usage of wrong fully-qualified method name.

6.3.7 Detection

In the detection step, we take inputs from our databases that contain canonical forms, constraints and fully qualified method names generated from the documentation. We also take the intermediate representations created from the target ML program where we want to detect the misuses.

The high-level detection algorithm is shown in Algorithm 1. The algorithm takes the new *code* and all the databases DB as input. M contains all the misuses returned from the program, N represents the abstract neural network (ANN), P represents the abstract ML pipeline and S represents the API sequence.

Algorithm 1 Misuse Detection

procedure DETECT ($code$,DB)
  $M \leftarrow M \cup getSingleMethodMisuse(code, DB)$
  $N \leftarrow getANN(code, DB)$
  $P \leftarrow getPipeline(code, DB)$
  $S \leftarrow getAPISequence(code, DB)$
  $M \leftarrow M \cup getMultiMethodMisuse(N, P, S, DB)$
  $M \leftarrow M \cup getIncorrectCallMisuse(code, DB)$
  $M \leftarrow M \cup symbolicAnalysis(N)$
  return M
end procedure
procedure GETANN ($code$, DB)
  $G \leftarrow$ get AST of code
  $S \leftarrow$ semantics of the APIs from DB
  $N \leftarrow$ empty ANN
  visit AST G
  for each node A in G **do**
    if A is a layer API **then**
      V $\leftarrow$ create node by extracting values/symbols for each keyword and argument, computing output dimension as value/symbol using the semantics of A from S
      $N \leftarrow N \cup$ edge(N, V)
    end if
  end for
  return N
end procedure
procedure GETSINGLEMETHODMISUSE($code$, DB)
  $M_s \leftarrow \emptyset$
  $C \leftarrow$ single method constraints from database
  $APIs \leftarrow$ API canonical form from code
  $V \leftarrow$ values of API arguments from code
  for each $A_c \in APIs$ **do**
    $m \leftarrow$ checkCanonicalForm(A_c, DB)
    if $m! = Null$ **then**
      $M_s.add(m)$
    end if
    $m \leftarrow$ checkParameterConstraint(A_c, C, V)
    if $m! = Null$ **then**
      $M_s.add(m)$
    end if
  end for
  return M_s
end procedure
procedure GETINCORRECTCALLMISUSE ($code$, DB)
  $M_f \leftarrow \emptyset$
  $F_c \leftarrow$ fully qualified API method names from $code$
  for each $fqn \in F_c$ **do**
    if fqn is not in DB **then**
      $M_f \leftarrow$ misuse(fqn)
    end if
  end for
  return M_f
end procedure
procedure SYMBOLICANALYSIS (N)
  $M_n \leftarrow$ symbolic analysis among the layers in N **return** M_n
end procedure

6.3.7.1 Single Method Call API Misuse detection

At line 2 of Algorithm 1, we detect all the single method call API misuses. It uses the API canonical form database, single-method call parameter constraint database, the canonical forms of the APIs generated from the *code* and the values of the parameters of the APIs generated from the *code*. From the canonical form database, we check whether the pattern of an API call violates all the accepted canonical forms. We iterate over all the canonical forms for an API and try to match by each key and value. If we do not find any pattern that matches the new call pattern, we detect a misuse. For example, consider the following code snippet:

```
model = Sequential()
model.add(Dense("relu"))
```

The canonical form created from the snippet above is `{"arg0": "String"}`. Our detector detects a misuse in the snippet as it violates all the canonical forms we mined for the `Dense` API. All the canonical forms for `Dense` API contains either `arg0` or `unit` which is integer. We check violation of good usage canonical form at line 28. For an API call A_c in the new code, if there is no correct canonical form in the DB we return a misuse message m otherwise we return $null$. Then, we add the misuse to our result at line 30. At line 32, we find the misuses for violation of single method call constraints. In this case, we utilize the single API constraints collected from *Stack Overflow*. First, we collect the values of different arguments used in an API A_c by visiting the AST of the program. Then, we get all the collected constraints for A_c from DB. After that, we iteratively verify if any of the constraints are violated. If any constraints is violated then we detect constraint violation misuse. The pseudocode of the process is shown in Algorithm 2.

Then, at line 34 of Algorithm 1, we add the misuses returned at line 32.

6.3.7.2 Multiple-method call misuse detection

At lines 3, 4 and 5 of Algorithm 1 we get the ANN, abstract ML pipeline and API sequences, respectively. At line 6, we use these artifacts to detect multiple-method call misuses. We get the multi-API constraints stored in the form of `<action, call_sequence>` from the APIs in *code*. We then iterate over

Algorithm 2 Detect Single API constraint violation

```
1: procedure CHECKPARAMETERCONSTRAINT (A_c, DB)
2:     m ← []
3:     C_s ← constraints for A_c from DB
4:     C_Ac ← get argument values in A_c
5:     for each c ∈ C_s do
6:         if C_Ac violates c then
7:             V_c ← violation message
8:             m.add(V_c)
9:         end if
10:    end for
11:    return m
12: end procedure
```

each of the multi API constraints and check whether there exists any constraint which is violated by the new usage. We report a misuse for each of the violations as described from lines 4 - 8 in Algorithm 3.

Algorithm 3 Detect Multi API constraint violation

```
1: procedure GETMULTIMETHODMISUSE(N, P, S, DB)
2:     M ← {}
3:     C ← multi method constraints from DB for each API in S
4:     for each constraint c ∈ C do
5:         if S violates c or P violated c or N violated c then
6:             M ← M ∪ m
7:         end if
8:     end for
       return M
9: end procedure
```

6.3.7.3 Detecting Missing ML Stage

To detect whether there are any missing stages in the ML program, we get the pipeline P as API call sequence of stages from the program *code* at line 4 of Algorithm 1. We describe the process of detecting misuse of missing ML stage in Algorithm 4. At line 3, we get the ideal pipeline from DB and detect the misuses from lines 4 - 14. We first we detect misuses that have missing required stage as m_c. Then we detect the misuses that have missing optional stages as m_p. These misuses have a risk of performance pitfall. Then

we detect the misuses where an order of the stages is violated as m_o. We add a misuse with a relevant message for each of these cases.

Algorithm 4 Detect Multi API constraint violation

```
1:  procedure DETECTMISSINGSTAGE( P, DB)
2:      M ← {}
3:      P_g ← get ideal pipeline from DB
4:      for each stage s_1, s_2 ∈ P_g do
5:          if any of s_1, s_2 is required and not present in P then
6:              M ← M ∪ m_c
7:          end if
8:          if any of s_1, s_2 is optional and not present in P then
9:              M ← M ∪ m_p
10:         end if
11:         if relative order of s_1, s_2 is violted in P then
12:             M ← M ∪ m_o
13:         end if
14:     end for
        return M
15: end procedure
```

For each violation we add a misuse and return the messages.

6.3.7.4 Detect Incorrect API call misuses

At line 7, we detect the incorrect API call misuses. We check the fully-qualified method names generated from the target code against the fully-qualified name database to detect misuses. For each of the APIs, we find whether it is attributed to wrong fully-qualified method name. If the API call fully-qualified name is absent in our DB we report a misuse. These misuses are found very often both in *Stack Overflow* and *Github* for *Keras* and *Tensorflow*.

6.3.7.5 Dimension mismatch misuse detection

At line 8, we use the ANN to check if the API usage causes a dimension mismatch. First, we get the ANN at line 3. In each node of the ANN, we get the dimension information for that layer. The ML APIs convert the dimension. We get this semantic information of conversion from API documentation and

Table 6.2: Detection accuracy.

		Detect		TP		Prec.		Recall		F_1	
Dataset	**Misuses**	***Proposed***	***BL***	***Proposed***	***BL***	***Proposed***	***BL***	***Proposed***	***BL***	***Proposed***	***BL***
Tensorflow SO	119	127	113	97	61	77%	54%	82%	52%	79%	53%
Keras SO	181	169	148	156	105	92%	71%	85%	57%	89%	63%
Tensorflow GH	101	117	87	84	59	72%	68%	83%	59%	77%	63%
Keras GH	354	385	289	301	230	78%	80%	85%	65%	82%	72%

Table 6.3: Detection results by ML pipeline stages.

	TF SO		Keras SO		TF GH		Keras GH		Overall	
	Total	Detected	Total	Detected	Total	Detected	Total	Detected	Total	Detected
Data Prep.	0	0	4	2	16	14	22	18	42	34
Modeling	72	62	125	115	64	56	294	256	555	489
Training	34	24	40	32	17	12	28	21	119	89
Evaluation	10	10	4	2	0	0	0	0	20	12
Tuning	3	1	3	2	4	2	10	6	20	11
Prediction	0	0	5	3	0	0	0	0	5	3

compute the ANN. If concrete dimensions are not available, we use symbolic values instead of concrete values. Then, we use z3py (a python library for Z3 [137] SMT solver) to run symbolic analysis to check if the constraints on the dimension between the layers is violated. If the violation is found we report a misuse with an appropriate message containing the misuse location.

Finally, we return all misuses (M) at line 8.

6.4 Empirical Evaluation

6.4.1 *Stack Overflow* (SO) Misuse Dataset

We extracted code snippets from *Stack Overflow* posts in the study. For each code snippet, we created an executable program and ran it locally to identify the misuses. To run the program we sometimes needed to import necessary libraries and create a mock dataset using `numpy` [138] library. A code snippet might have more than one misuses. We fixed misuses one-by-one until the program ran without errors and recorded all misuses. As described earlier here running without error means the training of the model finishes successfully without any training time error/exception and the prediction can also be made successfully. We do

Table 6.4: Detection results by program elements.

			TF SO	Keras SO	TF GH	Keras GH	Overall
Method Call	Single	Receiver		1|0		7|0	8|0
		Method	16|10	21|20	66|61	83|73	186|164
		Argument	62|53	139|123	35|23	259|225	492|424
	Multiple	Method Order	2|1				2|1
		Method Calls	29|28	14|11		4|2	47|41
		Argument	7|5	4|2		1|1	12|8
Iterations			3|0	1|0			4|0
Conditions				1|0			1|0

In each cell, the pair of values respectively indicate the numbers of total and detected missuses.

Table 6.5: Impact Analysis.

Impact	TF SO	Keras SO	TF GH	Keras GH	Total
Performance	18	16	9	123	166
Exception	60	77	28	51	216
Warning	41	88	64	180	373
Total	119	181	101	354	755

not consider the performance of the model in this process. The misuses identified through this process are only related to 'Exception' as shown in Table 6.5 , The number of misuses are shown in column `Misuses` of Table 6.2. The *Stack Overflow* dataset contains 181 misuses from *Keras* and 119 misuses from *Tensorflow*.

6.4.2 *Github* (GH) Misuse Dataset

We created this dataset from the misuse-fixing changes from repositories on *Github*. First, we used *Github* APIs to search for repositories that were written in *Python* and had keyword "Keras" or "Tensorflow" to collect all *Python* repositories relevant to *Keras* and *Tensorflow*, respectively. Then, in each repository, we used *Github* APIs to search for commits whose log messages contained keyword "fix". To exclude the commits which were not related to *Tensorflow* or *Keras* changes, we analyzed the code changes and only included ones containing *Tensorflow* or *Keras* APIs. For each library, we sort the repositories by their creation times in ascending order and manually analyzed the first 1,000 commits. We found 56 commits for *Keras* and 52 commits for *Tensorflow* that fixed ML API misuses. Then, we collected both buggy and fixed code snippets from the commits and added them to the *Github* dataset. The number of misuses are shown in

column `Misuses` of Table 6.2. The *Github* dataset contains 354 misuses from *Keras* and 101 misuses from *Tensorflow*.

6.4.3 Accuracy Metrics

We manually created an executable programs for the detected misuses and ran them to validate the misuses. We counted the number of validated misuses as TP. Then, we computed the **precision** and the **recall** as the ratio between TP over the number of detected misuses and the number of expected misuses in the dataset, respectively. We also computed the harmonic average of the precision and the recall $F_1 = 2 \times Precision \times Recall/(Precision + Recall)$.

6.4.4 ML API Misuse Detection Accuracy

We first ran our tool on the *Stack Overflow* misuses dataset to evaluate its performance on the same dataset used to draw the observations and develop the solution. Then, we ran our tool on *Github* misuses dataset which is a new dataset independent of the dataset used in the study to evaluate if the approach generalizes to other datasets.

Baseline Approach (BL) We compare our approach with [70] referred as BL in Table 6.2. BL uses the Boa infrastructure [112, 113] to extract API usage sequences for a fixed set of APIs (100 APIs in [70]) from a large number of repositories (over 380K GitHub). Then, it performs slicing to remove irrelevant statements. Finally, frequent subsequence mining and guard condition mining is used to extract temporal ordering of related API calls. BL is currently the best known API misuse detection approach.

Table 6.2 shows the detection accuracy on both *Stack Overflow* and *Github* datasets. We achieve high accuracy on both datasets and for both libraries. The precision is higher than 70% and the recall is higher than 80%. Compared to BL, our approach detects dimensionality related API misuses that we detect using symbolic analysis, misuses due to value combination of arguments, optional hyperparameters, etc.

F_1 scores are comparable between the two datasets which shows that the design of our approach based on the observations from *Stack Overflow* dataset could be generalized to detect ML API misuses in other datasets.

6.4.5 Analyzing Classification of Detection Results

We further analyze the characteristics of the misuses by mapping them to the classification.

Table 6.3 shows the classification of the misuses by ML pipeline stages. The result for the *Github* dataset is consistent with the *Stack Overflow* dataset. Most misuses are in training and modeling. However, in the *Github* dataset, modeling misuses are dominant. This is because similar posts are flagged on *Stack Overflow*, while multiple *Github* projects might contain similar mistakes. In terms of detection capability, our approach performed equally well on all stages.

Table 6.4 shows the classification by roles of API elements in the misuses. These results are consistent between *Github* and *Stack Overflow* datasets. Overall, while our approach detects misuses with both single and multiple method calls quite well, there were fewer misuses involving conditions and iterations. Furthermore, our approach worked well for methods and arguments.

To understand the impact of misuses, we created an executable program for each misuse and ran it, and record its impact. We classified impacts into three classes: performance, exception and warning. Overall, we see misuses with all different classes of impacts. The warning is the most frequent, then exception and finally performance. However, this trend is not consistent for all datasets or libraries.

6.4.6 Usefulness Analysis

To demonstrate the usefulness of our tool, we ran it on the last snapshots of open source projects which use *Tensorflow* or *Keras*, and are active on *Github* to detect not-yet-known misuses. We consider a project active if it has committed in the last two weeks. As a result, we got 20 projects for *Tensorflow* and 6 for *Keras*. Table 6.6 shows the result of running our tool on the latest versions of those projects. Among 67 and 71 detected *Tensorflow* and *Keras* API misuses, we manually checked and found that 25 and 27 are true misuses, respectively. That is precision of 37% and 38%. One of the authors created 20 pull requests with fixes of those misuses and at the time of this submission, we got seven responses from the developers. A developer from each of the seven projects confirmed the reported misuse. The precision in the wild projects was low as our approach reported many false positives in big projects with modularized structure and overriding the library API to create a customized version. The custom APIs were created in

Table 6.6: Not-yet-known misuse detection result.

Library	Repo.	Detect	TP	Prec.	Resp.	Confirm
Tensorflow	20	67	25	37%	5	5
Keras	6	71	27	38%	2	2

these projects which have the same name as the library API but have totally different signature. In many cases the pipeline stages are implemented in different files or modules, multi API constraints are satisfied by modularizing into different files and modules. Our approach reported a number of these cases as misuses yielding higher false positives.Intra module and intra file analysis support is left as future work. Currently, the response rate of the PRs is low as we have noticed most of the projects are by single developer who are not responsive or unaware of the pull requests. We have got 100% response rate from the PRs submitted to the projects of two industry leaders Google and IBM.

6.5 Conclusion

In this chapter, we presented an empirical study of ML API misuses followed by a detection strategy to detect the ML specific API misuses. Our empirical study on *Tensorflow* and *Keras Stack Overflow* posts shows respectively 6% and 13% of the studied posts contain misuses indicating the significant existence of API misuses in ML programming.

The study also motivated us to develop Amimla, an automatic API misuse detection technique and tool that can help developers to find the misuses in the ML programs. Amimla is found to detect the misuses from *Stack Overflow* and *Github* misuse datasets successfully showing more than 70% of F1 score. It is found to be useful in unseen datasets to detect misuses some of which have already been confirmed by the developers.

CHAPTER 7. CONCLUSION AND FUTURE WORK

The increasing usage of ML in developing software has opened new research challenges for the researchers. In this thesis, we explore some challenges faced by the developers which need immediate attention from both the researchers and the API developers. In Chapter 3, we have presented a large scale study of the difficulties faced by the developers. Key findings in Chapter 3 show that developers are likely to face challenges primarily in data preparation, model creation, and training stages of the ML pipeline. All these findings point out to the development of new principles and paradigms for ML API developments and different ML specific static analysis tools. In Chapter 4, we have conducted a comprehensive study to understand the bug patterns in developing DNN software. We have found that there are some unique bug patterns like IPS, Data flow bug, and model creation bugs which are different from other domains of software engineering. This implies that we need new kinds of debuggers and bug detectors to aid the development of DNN applications. In Chapter 5, we have presented another empirical study on Fixing DNN by the developers. We have found some new patterns of fixing DNN models like fixing network connection, fixing data dimension, fixing layer dimension, etc which are new and unique for DNNs. The fix patterns can be used to automate the repairing of DNN models. In Chapter 6, we proposed AMIMLA, an automatic technique to detect API misuses made by the developers. We have shown that our approach outperforms the state of the art approaches in other domains of software engineering in finding ML misuses.

7.1 Future Work

The contributions made in this thesis can lead to many potential research directions. Some of these future research directions are discussed below:

7.1.1 ML Pipeline Design

We have shown that the current approach of designing ML pipeline causes problems among the developers. Developers are having difficulty in creating error-free pipelines using the ML libraries. The pipelines fail without proper explanation or reasoning, sometimes the performance is bad out of the pipeline, sometimes one of the stages becomes a bottleneck for the whole pipeline. So it is an urgent call for the researchers to think about ideal strategies and principles on designing a more explainable and error-free ML pipeline. We need both theoretical and applied techniques in pipeline design like the design patterns in software development.

7.1.2 ML Specific Static and Dynamic Analysis Tools

Current static and dynamic analysis tools have been found not suitable for ML. However, both static and dynamic analysis tools have the potential to identify the ML specific problems from software and produce more robust ML software. This is a critical need as we need to enhance our trust in ML models before it can be brought to production for safety-critical systems. The findings and results presented in this thesis can aid both researchers and practitioners to develop novel ML specific static and dynamic analysis tools.

7.1.3 ML API Design and Abstraction

We have shown that the current strategy of API design is not suitable for ML. A highly abstract ML API is currently being served as a black box. So the developer does not know how the model is learning, when is it failing, etc. So we need to take a step back and rethink the principles of API design for ML. This remains an immediate future work for the researchers.

7.1.4 ML Specific Debuggers

Currently, ML software is developed in the IDEs which contain the debugging support for traditional software. However, these tools are not sufficient for debugging ML programs. Google has made some attempts to debug DNN models through TFDebug [129], however its scope is currently very limited. So, we need immediate attention from researchers to design ML specific debuggers.

7.1.5 Automatic Detection of ML Bugs

We have discussed an automatic API misuse detection strategy in our thesis. We have shown that the detection strategy needs new technical innovations for ML. Similarly, we need techniques to detect other ML specific bugs as discussed in our thesis.

7.1.6 Automatic Program Repair

Our thesis identifies some unique patterns of fixing DNN bugs. We have discussed that these patterns are not present in other domains of software engineering. Therefore, it is a new research challenge for the researchers to design automatic ML program repair tools that can be built on top of the patterns we identify.

7.1.7 ML Specific Intermediate Representation (IR) for Enabling ML Program Analysis

We have been able to show that DNN libraries show a similar pattern of bugs. Similarly, non-DNN libraries also show a similar kind of difficulties. This motivates the possibility of whether we can build a common intermediate representation (IR) for the analyses of ML programs. For example, instead of having five different program analyses tools for five different DNN libraries, we can have a common program analysis tool. The input program from different libraries will be first converted to a common IR and the program analyses will be done on the IR.

7.1.8 Software Testing for ML Software

An immediate research challenge that evolves from our work is related to software testing of ML software. It is an urgent call to design new theories on testing ML software to enable proper testing. We also need to understand whether the current testing strategies are suitable in ML or not.

7.1.9 Reuse of ML Models and Versioning

We have shown that many difficulties and bugs originate while reusing the existing DNN models. So we need to understand what would be an ideal strategy to enable model reuse to reduce the work from scratch. We need to as we understand whether the current versioning strategies using Github are sufficient for ML models or not. It remains a challenge to both the researchers and practitioners.

BIBLIOGRAPHY

[1] Yufeng Guo, "The 7 Steps of Machine Learning," 2017. https://towardsdatascience.com/the-7-steps-of-machine-learning-2877d7e5548e.

[2] M. J. Islam, H. A. Nguyen, R. Pan, and H. Rajan, "What do developers ask about ML libraries? a large-scale study using stack overflow," *arXiv preprint arXiv:1906.11940*, 2019.

[3] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), pp. 510–520, ACM, 2019.

[4] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *ICSE'20: The 42nd International Conference on Software Engineering*, May 2020.

[5] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 422–431, IEEE Press, 2013.

[6] M. J. Islam, A. Sharma, and H. Rajan, "A cyberinfrastructure for big data transportation engineering," *Journal of Big Data Analytics in Transportation*, vol. 1, no. 1, pp. 83–94, 2019.

[7] S. Biswas, M. J. Islam, Y. Huang, and H. Rajan, "Boa meets python: A boa dataset of data science software in python language," in *MSR'19: 16th International Conference on Mining Software Repositories*, May 2019.

[8] R. Pan, M. J. Islam, S. Ahmed, and H. Rajan, "Identifying classes susceptible to adversarial attacks," *arXiv preprint arXiv:1905.13284*, 2019.

[9] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, ACM, 2014.

[10] A. Candel, V. Parmar, E. LeDell, and A. Arora, "Deep learning with h2o," *H2O. ai Inc*, 2016.

[11] F. Chollet *et al.*, "Keras." https://github.com/fchollet/keras, 2015.

[12] S. Owen and S. Owen, *Mahout in action*. Manning Shelter Island, NY, 2012.

[13] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning.," in *OSDI*, vol. 16, pp. 265–283, 2016.

[16] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron, *et al.*, "Theano: Deep learning on gpus with python," in *NIPS 2011, BigLearning Workshop, Granada, Spain*, vol. 3, Citeseer, 2011.

[17] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," tech. rep., Idiap, 2002.

[18] G. Holmes, A. Donkin, and I. H. Witten, "Weka: A machine learning workbench," in *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pp. 357–361, IEEE, 1994.

[19] S. Meldrum, S. A. Licorish, and B. T. R. Savarimuthu, "Crowdsourced knowledge on stack overflow: A systematic mapping study," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pp. 180–185, ACM, 2017.

[20] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?: Nier track," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 804–807, IEEE, 2011.

[21] D. Kavaler, D. Posnett, C. Gibler, H. Chen, P. Devanbu, and V. Filkov, "Using and asking: APIs used in the android market and asked about in stackoverflow," in *International Conference on Social Informatics*, pp. 405–418, Springer, 2013.

[22] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do API changes trigger stack overflow discussions? a study on the android sdk," in *proceedings of the 22nd International Conference on Program Comprehension*, pp. 83–94, ACM, 2014.

[23] J. L. Berral-García, "A quick view on current techniques and machine learning algorithms for big data analytics," in *2016 18th international conference on transparent optical networks (ICTON)*, pp. 1–4, IEEE, 2016.

[24] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical Software Engineering*, vol. 19, no. 3, pp. 619–654, 2014.

[25] M. Rebouças, G. Pinto, F. Ebert, W. Torres, A. Serebrenik, and F. Castor, "An empirical study on the usage of the swift programming language," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1, pp. 634–638, IEEE, 2016.

[26] D. Schenk and M. Lungu, "Geo-locating the knowledge transfer in stackoverflow," in *Proceedings of the 2013 International Workshop on Social Software Engineering*, pp. 21–24, ACM, 2013.

[27] C. Stanley and M. D. Byrne, "Predicting tags for stackoverflow posts," in *Proceedings of ICCM*, vol. 2013, 2013.

[28] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 70–79, IEEE, 2013.

[29] A. Baltadzhieva and G. Chrupała, "Predicting the quality of questions on stackoverflow," in *Proceedings of the International Conference Recent Advances in Natural Language Processing*, pp. 32–40, 2015.

[30] A. Joorabchi, M. English, and A. E. Mahdi, "Text mining stackoverflow: An insight into challenges and subject-related difficulties faced by computer science learners," *Journal of Enterprise Information Management*, vol. 29, no. 2, pp. 255–275, 2016.

[31] D. Sculley, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Machine learning: The high-interest credit card of technical debt," 2014.

[32] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, (Cambridge, MA, USA), pp. 2503–2511, MIT Press, 2015.

[33] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "What's your ML test score? a rubric for ML production systems," in *NIPS Workshop on Reliable Machine Learning in the Wild*, 2016.

[34] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: a case study," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, pp. 291–300, IEEE Press, 2019.

[35] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 329–339, ACM, 2008.

[36] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proceedings of the 2018 26th ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 539–550, ACM, 2018.

[37] G. Bavota, M. Linares-Vasquez, C. E. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change-and fault-proneness on the user ratings of android apps," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, 2015.

[38] D. Dig and R. Johnson, "How do API evolve? a story of refactoring," *Journal of software maintenance and evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[39] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How does web service API evolution affect clients?," in *2013 IEEE 20th International Conference on Web Services*, pp. 300–307, IEEE, 2013.

[40] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "An empirical study on the impact of refactoring activities on evolving client-used APIs," *Information and Software Technology*, vol. 93, pp. 186–199, 2018.

[41] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[42] R. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. Prasad, "Bugs. jar: A large-scale, diverse dataset of real-world Java bugs," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 10–13, IEEE, 2018.

[43] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in Java programs," *Journal of Systems and Software*, vol. 106, pp. 82–101, 2015.

[44] W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu, "How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 381–392, IEEE, 2017.

[45] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*, pp. 93–102, ACM, 2011.

[46] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pp. 25–33, ACM, 2006.

[47] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *ACM SIGOPS Operating Systems Review*, vol. 35, pp. 73–88, ACM, 2001.

[48] J. Park, M. Kim, B. Ray, and D.-H. Bae, "An empirical study of supplementary bug fixes," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pp. 40–49, IEEE Press, 2012.

[49] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 97–106, ACM, 2010.

[50] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 913–923, IEEE Press, 2015.

[51] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 129–140, ACM, 2018.

[52] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pp. 271–280, IEEE, 2012.

[53] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *2013 IEEE International Conference on Software Maintenance*, pp. 270–279, IEEE, 2013.

[54] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li, "An empirical study on real bugs for machine learning programs," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 348–357, IEEE, 2017.

[55] T. Zhang, C. Gao, L. Ma, M. R. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2019.

[56] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *Proceedings of the 41st International Conference on Software Engineering*, pp. 1027–1038, IEEE Press, 2019.

[57] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, pp. 943–956, Nov. 1992.

[58] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static API-misuse detectors," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[59] B. Ganter and R. Wille, *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, 2012.

[60] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: lightweight cross-project anomaly detection," in *Proceedings of the 19th international symposium on Software testing and analysis*, pp. 119–130, ACM, 2010.

[61] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," *ACM Sigplan Notices*, vol. 37, no. 1, pp. 4–16, 2002.

[62] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[63] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 15–24, ACM, 2010.

[64] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 371–382, IEEE Computer Society, 2009.

[65] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 925–935, IEEE Press, 2012.

[66] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 35–44, ACM, 2007.

[67] Z. Li and Y. Zhou, "Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, (New York, NY, USA), pp. 306–315, ACM, 2005.

[68] M. Monperrus, M. Bruch, and M. Mezini, "Detecting missing method calls in object-oriented software," in *European Conference on Object-Oriented Programming*, pp. 2–25, Springer, 2010.

[69] S. Thummalapenta and T. Xie, "Alattin: mining alternative patterns for defect detection," *Automated Software Engineering*, vol. 18, no. 3-4, pp. 293–323, 2011.

[70] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 886–896, IEEE, 2018.

[71] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, (New York, NY, USA), pp. 383–392, ACM, 2009.

[72] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 109–119, ACM, 2018.

[73] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *International Conference on Computer Aided Verification*, pp. 3–29, Springer, 2017.

[74] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*, pp. 97–117, Springer, 2017.

[75] W. Ruan, X. Huang, and M. Kwiatkowska, "Reachability analysis of deep neural networks with provable guarantees," *arXiv preprint arXiv:1805.02242*, 2018.

[76] kdnuggets, "Top 15 Frameworks for Machine Learning Experts," 2016. https://www.kdnuggets.com/2016/04/top-15-frameworks-machine-learning-experts.html.

[77] S. Wang, D. Lo, and L. Jiang, "An empirical study on developer interactions in stackoverflow," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 1019–1024, ACM, 2013.

[78] J. Yang, K. Tao, A. Bozzon, and G.-J. Houben, "Sparrows and owls: Characterisation of expert behaviour in stackoverflow," in *International Conference on User Modeling, Adaptation, and Personalization*, pp. 266–277, Springer, 2014.

[79] T. P. Sahu, N. K. Nagwani, and S. Verma, "Selecting best answer: An empirical analysis on community question answering sites," *IEEE Access*, vol. 4, pp. 4797–4808, 2016.

[80] W. Wang and M. W. Godfrey, "Detecting api usage obstacles: A study of ios and android developer questions," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 61–64, IEEE Press, 2013.

[81] Y. Zhang, G. Yin, T. Wang, Y. Yu, and H. Wang, "Evaluating bug severity using crowd-based knowledge: An exploratory study," in *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, pp. 70–73, ACM, 2015.

[82] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

[83] Z. Wang, K. Liu, J. Li, Y. Zhu, and Y. Zhang, "Various frameworks and libraries of machine learning and deep learning: A survey," *Archives of Computational Methods in Engineering*, pp. 1–24, 2019.

[84] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. L. García, I. Heredia, P. Malík, and L. Hluchỳ, "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey," *Artificial Intelligence Review*, pp. 1–48, 2019.

[85] S. Lockyer, "Coding qualitative data," *The Sage encyclopedia of social science research methods*, vol. 1, no. 1, pp. 137–138, 2004.

[86] R. S. Life, "Qualitative data analysis," 1994.

[87] A. Strauss and J. Corbin, *Basics of qualitative research*. Sage publications, 1990.

[88] A. Radu and S. Nadi, "A dataset of non-functional bugs," in *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, (Piscataway, NJ, USA), pp. 399–403, IEEE Press, 2019.

[89] P. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, pp. 78–87, Oct. 2012.

[90] K. A. Hallgren, "Computing inter-rater reliability for observational data: an overview and tutorial," *Tutorials in quantitative methods for psychology*, vol. 8, no. 1, p. 23, 2012.

[91] G. Guzzetta, G. Jurman, and C. Furlanello, "A machine learning pipeline for quantitative phenotype prediction from genotype data," *BMC bioinformatics*, vol. 11, no. 8, p. S3, 2010.

[92] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1192–1223, 2016.

[93] Stack Overflow, "Stack Overflow Privilege Policy," 2019. https://stackoverflow.com/help/privileges.

[94] A. Said and A. Bellogín, "Comparative recommender system evaluation: benchmarking recommendation frameworks," in *Proceedings of the 8th ACM Conference on Recommender systems*, pp. 129–136, ACM, 2014.

[95] H. Jin, Q. Song, and X. Hu, "Auto-keras: Efficient neural architecture search with network morphism," *arXiv preprint arXiv:1806.10282*, 2018.

[96] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 847–855, ACM, 2013.

[97] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 263–272, ACM, 2005.

[98] Alexandr Honchar, "Using Caffe with your own dataset," 2017. https://medium.com/machine-learning-world/using-caffe-with-your-own-dataset-b0ade5d71233.

[99] A. Chakarov, A. Nori, S. Rajamani, S. Sen, and D. Vijaykeerthy, "Debugging machine learning tasks," *arXiv preprint arXiv:1603.07292*, 2016.

[100] Tensorflow, "Debugging TensorFlow Programs," 2016. https://www.tensorflow.org/programmers_guide/debugger.

[101] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," *arXiv preprint arXiv:1711.00489*, 2017.

[102] I. Goodfellow, "Efficient per-example gradient computations," *arXiv preprint arXiv:1510.01799*, 2015.

[103] S. Saito and S. Roy, "Effects of loss functions and target representations on adversarial robustness," *arXiv preprint arXiv:1812.00181*, 2018.

[104] D. Krstajic, L. J. Buturovic, D. E. Leahy, and S. Thomas, "Cross-validation pitfalls when selecting and assessing regression and classification models," *Journal of cheminformatics*, vol. 6, no. 1, p. 10, 2014.

[105] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: A benchmark for API-misuse detectors," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, (New York, NY, USA), pp. 464–467, ACM, 2016.

[106] H. W. Lilliefors, "On the kolmogorov-smirnov test for normality with mean and variance unknown," *Journal of the American statistical Association*, vol. 62, no. 318, pp. 399–402, 1967.

[107] T. T. D. Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, 2016.

[108] B. Beizer, *Software system testing and quality assurance*. Van Nostrand Reinhold Co., 1984.

[109] A. J. Viera, J. M. Garrett, *et al.*, "Understanding interobserver agreement: the kappa statistic," *Fam med*, vol. 37, no. 5, pp. 360–363, 2005.

[110] D. Gómez and A. Rojas, "An empirical overview of the no free lunch theorem and its effect on real-world machine learning classification," *Neural computation*, vol. 28, no. 1, pp. 216–228, 2016.

[111] Alexander Shvets, "Software Development AntiPatterns," 2017. https://sourcemaking.com/antipatterns/software-development-antipatterns.

[112] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *35th International Conference on Software Engineering*, ICSE'13, pp. 422–431, May 2013.

[113] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: Ultra-large-scale software repository and source-code mining," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 7:1–7:34, 2015.

[114] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: an enabling language and infrastructure for ultra-large scale msr studies," *The Art and Science of Analyzing Software Data*, pp. 593–621, 2015.

[115] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[116] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "A benchmark for bugs and fix patterns for deep neural networks," 2020. https://github.com/lab-design/ICSE2020DNNBugRepair.

[117] C. Xie, J. Wang, Z. Zhang, Z. Ren, and A. Yuille, "Mitigating adversarial effects through randomization," *arXiv preprint arXiv:1711.01991*, 2017.

[118] Q. Xiao, K. Li, D. Zhang, and Y. Jin, "Wolf in sheep's clothing-the downscaling attack against deep learning applications," *arXiv preprint arXiv:1712.07805*, 2017.

[119] J. H. Friedman, "On bias, variance, 0/1 loss, and the curse-of-dimensionality," *Data mining and knowledge discovery*, vol. 1, no. 1, pp. 55–77, 1997.

[120] I. Jolliffe, *Principal component analysis*. Springer, 2011.

[121] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[122] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[123] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *arXiv preprint arXiv:1702.05659*, 2017.

[124] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 64–73, IEEE, 2014.

[125] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Why and how Java developers break APIs," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 255–265, IEEE, 2018.

[126] K. Jezek, J. Dietrich, and P. Brada, "How Java API break–an empirical study," *Information and Software Technology*, vol. 65, pp. 129–146, 2015.

[127] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.

[128] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," *arXiv preprint arXiv:1206.6389*, 2012.

[129] TensorFlow, "TensorFlow Github," 2019. https://github.com/tensorflow/tensorflow/tags/.

[130] G. Uddin, B. Dagenais, and M. P. Robillard, "Temporal analysis of API usage concepts," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 804–814, IEEE Press, 2012.

[131] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of APIs in large-scale code corpus," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 166–177, ACM, 2014.

[132] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 39–57, IEEE, 2017.

[133] H. Rajan, "D4 (dependable data driven discovery) framework," tech. rep., Iowa State University, Feb. 2020.

[134] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *ESEC/FSE'19: The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE 2019, Aug. 2019.

[135] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming q&a in stackoverflow," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 25–34, IEEE, 2012.

[136] A. Di Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 509–519, IEEE, 2017.

[137] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[138] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.